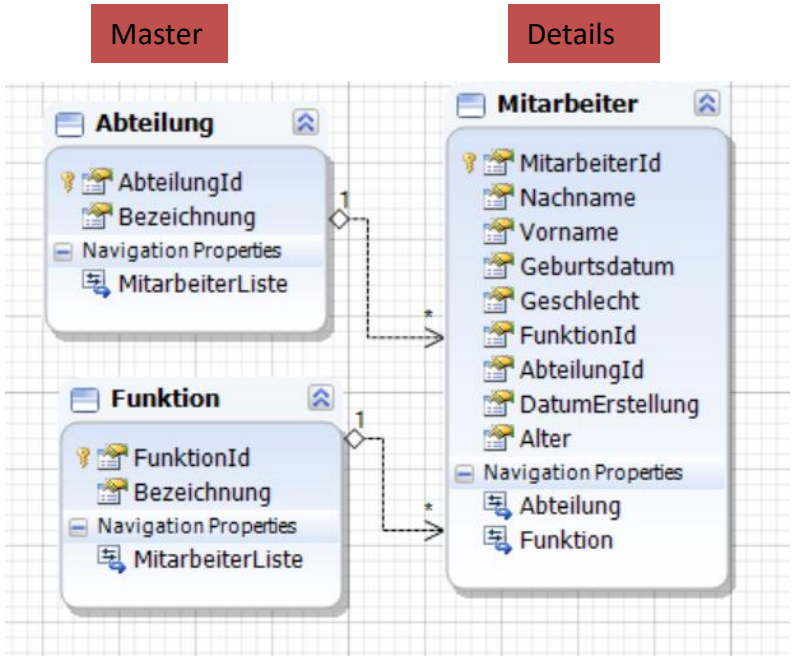
Die beiden Bestandteile des EF Core-Modells

1. Die Modell- oder Entitätsklassen (POCO)
2. Die aus EntityFrameworkCore.DBContext abgeleitete eigene Klasse <projektname>DBContext zur Verbindung mit der Datenbank

Das EF Core- Modell
die Modell-oder Entitätsklassen

- Eine Anwendung mit Datenbank-Persistenzlösung per EF Core besitzt
  - Klassen zur Modellierung der sogenannten Entitäten aus dem Aufgabenbereich. Eine Entitätsklasse ist in der Regel mit einer Tabelle einer relationalen Datenbank assoziiert. Beispiel:



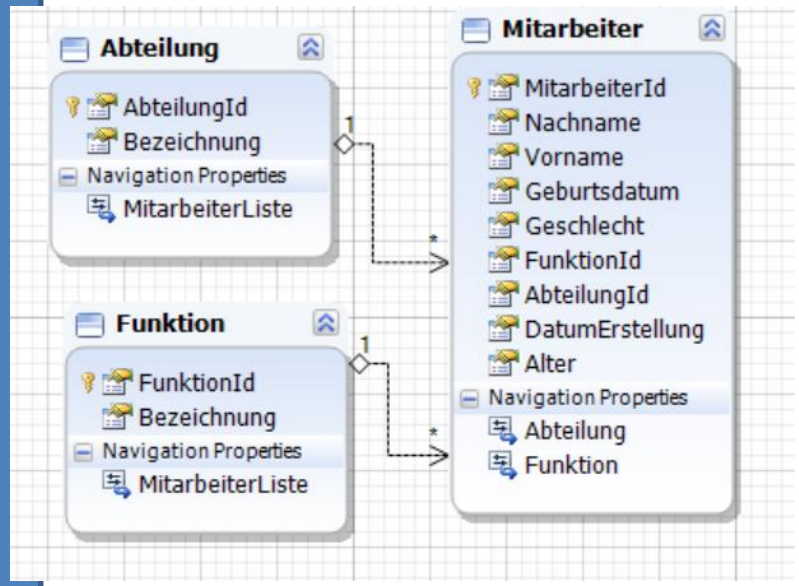

```
public class Abteilung
{
  public int idAbteilung { get; set; }
  public string Abteilungsbezeichnung { get; set; }

  public List <Mitarbeiter> MitarbeiterListe{ get; }
                = new List<Mitarbeiter>();          // NavigationProperty
}
```

```
public class Mitarbeiter
{
  public int idMitarbeiter { get; set; }
  public string Nachname { get; set; }
....
  public Abteilung Abteilung { get; set; }          // NavigationProperty
  public Funktion Funktion { get; set; }
}
```

Das EF Core- Modell – 1. die Modell- oder Entitätsklassen.
Durch Migration (Forward Engineering) werden die Tabellen in der Datenbank generiert

- Eine Anwendung mit Datenbank-Persistenzlösung per EF Core besitzt
  - Klassen zur Modellierung der sogenannten Entitäten aus dem Aufgabenbereich. Eine Entitätsklasse ist in der Regel mit einer Tabelle einer relationalen Datenbank assoziiert. Beispiel:




```
public class Abteilung
{
  public int idAbteilung { get; set; }
  public string Bezeichnung { get; set; }

  public List <Mitarbeiter> MitarbeiterListe{ get; }
                = new List();}     // NavigationProperty
```

```
public class Mitarbeiter
{
  public int idMitarbeiter { get; set; }
  public string Nachname { get; set; }
....                                  // NavigationProperty
  public Abteilung Abteilung { get; set; }
  public Funktion Funktion { get; set; }
}
```

	Spaltenname	Datentyp	NULL-Werte zulassen
🔑	abteilungId	int	☐
▶	bezeichnung	varchar(45)	☑

	Spaltenname	Datentyp	NULL-Werte zulassen
🔑	MitarbeiterId	int	☐
	Nachname	nvarchar(50)	☐
	Vorname	nvarchar(50)	☑
	Geburtsdatum	date	☑
	Geschlecht	char(1)	☑
	FunktionId	int	☐
	AbteilungId	int	☐
	DatumErstellung	date	☑
	[Alter]	int	☑

Migration:
Forward Engineering

Anforderungen an die Modell- bzw. Entitätsklassen

- Wie in den Beispielen gezeigt, handelt es sich bei den Modellklassen um so genannte POCOs (Plain Old Class Object bzw. Plain Old CLR Object), d.h. Klassen ohne Konstruktoren (der einfache Konstruktor ohne Parameter ist aber erlaubt), ohne Vererbung, ohne Verhalten, Properties auf der Basis einfacher (primitiver) Datentypen – eine Klasse, deren Objekte ausschließlich zur Datenspeicherung/übertragung dienen
- Die Abbildung der Modellklassen-Strukturen in ein Datenbank-Schema wird im Laufe des Kurses mehr und mehr vertieft und erfolgt auf der Basis
  - Vorhandener Konventionen
  - So genannter Annotationen (Attribute, die den Properties zugewiesen werden)
  - Mit Hilfe von Fluid API-Methoden in der DBContext-Klasse
- Im Folgenden werden auch typische objektorientierte Aspekte (Assoziation, Vererbung, Erweiterung des Spektrums der Property-Datentypen, Methoden) diskutiert – erst dann werden wir die Vorteile der beiden Konzepte (Objektorientiert vs. relational) vereinen und richtig nutzen können

Das EF Core- Modell:
2. Die aus der Klasse DBContext abgeleitete eigene Kontext-Klasse

- Eine Anwendung mit Datenbank-Persistenzlösung per EF Core besitzt ein Modell mit
  - Klassen zur Modellierung der sogenannten Entitäten aus dem Aufgabenbereich. Eine Entitätsklasse ist in der Regel mit einer Tabelle einer relationalen Datenbank assoziiert. → siehe vorhergehende Folie

  und

  - einer von EntitiyFrameworkCore.DbContext abgeleiteten Klasse zur ORM-Vermittlung zwischen den Entitäten und der Datenbank. Ein Objekt der Kontextklasse u.a. dafür zuständig,
    - Die Verbindung zur Datenbank zu managen,
    - (das Modell und die Beziehungen zwischen den Klassen zu konfigurieren)
    - Tabellenzeilen der Datenbank abzufragen und daraus Objekte einer Entitätsklasse zu erstellen,
    - (Änderungen der Entitätsobjekte zu verfolgen)
    - modifizierte Entitätsobjekte in die Datenbank zu sichern.

```
public class EFC01DBContext : DbContext //Entity.Framework.Core
{
public EFC01DBContext() {}
public EFC01DBContext(DbContextOptions<EFC01DBContext> options)
                                                    : base(options) {}

   public DbSet<Abteilung> AbteilungListe { get; set; }
   public DbSet<Funktion>  FunktionListe { get; set; }
   public DbSet <Mitarbeiter> MItarbeiterListe{ get; set; }

   protected override void OnConfiguring(DbContextOptionsBuilder options)
   {
       options.UseSqlServer(sqlConnectionString);
   }

   protected override void OnModelCreating(ModelBuilder modelBuilder)
   {
       ....
   }
}
```

Wichtige Methoden der Klasse DBContext

Methode	Beschreibung
Add/AddAsync, AddRange	Ermöglicht das Einfügen der Entität in die Datenbank; beginnt die Verfolgung der Entität. Hinzugefügte Objekte werden nachverfolgt und hinzugefügt, sobald die Methode SaveChanges aufgerufen wird. Es gibt auch eine AddRange-Methode, die das gleichzeitige Hinzufügen von mehr als einer Entität ermöglicht.
Find/FindAsync	Findet eine spezifische Entity mit Hilfe des Primärschlüssel-Wertes (im Allg. Id).
OnConfiguring	Ermöglicht es uns, die Optionen und andere Informationen über die Datenbank zu überschreiben.
OnModelCreating	Ermöglicht die Verwendung der FluentAPI, um unsere Entitäten und ihre Beziehungen weiter zu definieren, indem die Modelle, ihre Eigenschaften und alle Beziehungen konfiguriert werden. Bei Bedarf kann auf Testdaten (Seed) zugegriffen werden
Remove, RemoveRange	Löscht eine Entity aus der Datenbank. (auch RemoveRange)
SaveChanges/Save ChangesAsync	Anwendung der verfolgten Änderungen in einer einzigen Transaktion
Update UpdateRange	Wird verwendet, um eine Aktualisierung der verfolgten Entität durchzuführen. Es besteht auch die Möglichkeit, einen Bereich von verfolgten Entitäten mit UpdateRange zu aktualisieren

Weitere Methoden der Klasse DBContext

Methode	Beschreibung
Entry<TEntity>(TEntity)	Ruft ein DbEntityEntry<TEntity> -Objekt für die angegebene Entität ab, die Zugriff auf Informationen über die Entität und die Möglichkeit zum Ausführen von Aktionen für die Entität bietet.
Set<TEntity>()	Gibt einen DbSet<TEntity> instance für den Zugriff auf Entitäten des angegebenen Typs im Kontext und im zugrunde liegenden Speicher zurück.

Die Methode OnConfiguring in der DBContext-Klasse

- Die wichtigste Aufgabe dieser Methode ist es, den Zugriff auf eine Datenbank zu ermöglichen. Aktuell stehen folgende Provider zur Verfügung

Database system	Example configuration	NuGet package
SQL Server or Azure SQL	`.UseSqlServer(connectionString)`	Microsoft.EntityFrameworkCore.SqlServer
Azure Cosmos DB	`.UseCosmos(connectionString, databaseName)`	Microsoft.EntityFrameworkCore.Cosmos
SQLite	`.UseSqlite(connectionString)`	Microsoft.EntityFrameworkCore.Sqlite
EF Core in-memory database	`.UseInMemoryDatabase(databaseName)`	Microsoft.EntityFrameworkCore.InMemory
PostgreSQL*	`.UseNpgsql(connectionString)`	Npgsql.EntityFrameworkCore.PostgreSQL
MySQL/MariaDB*	`.UseMySql(connectionString)`	Pomelo.EntityFrameworkCore.MySql
Oracle*	`.UseOracle(connectionString)`	Oracle.EntityFrameworkCore

Aufgaben der DBContext-Klasse in der Anwendung

- In der Regel dient ein DbContext-Objekt nicht zur Versorgung einer kompletten Anwendung, sondern es verwaltet eine Arbeitseinheit mit Datenbankbeteiligung, die eine kurze zeitliche Ausdehnung besitzt und typischerweise die folgenden Schritte umfasst:
  - Es wird ein DbContext-Objekt erstellt.
  - Es werden Entitäten aus der Datenbank abgefragt oder neu erstellt.
  - An den Entitäten werden von der Geschäftslogik diktierte Änderungen vorgenommen, die das DbContext-Objekt nachverfolgt.
  - Durch die DbContext-Methoden SaveChanges() oder SaveChangesAsync() werden registrierte Änderungen an Entitätsobjekten in die Datenbank geschrieben.
  - Das DbContext-Objekt erhält einen Dispose() – Aufruf, wird also automatisch aus dem Speicher entfernt nach der letzten geschweiften Klammer:

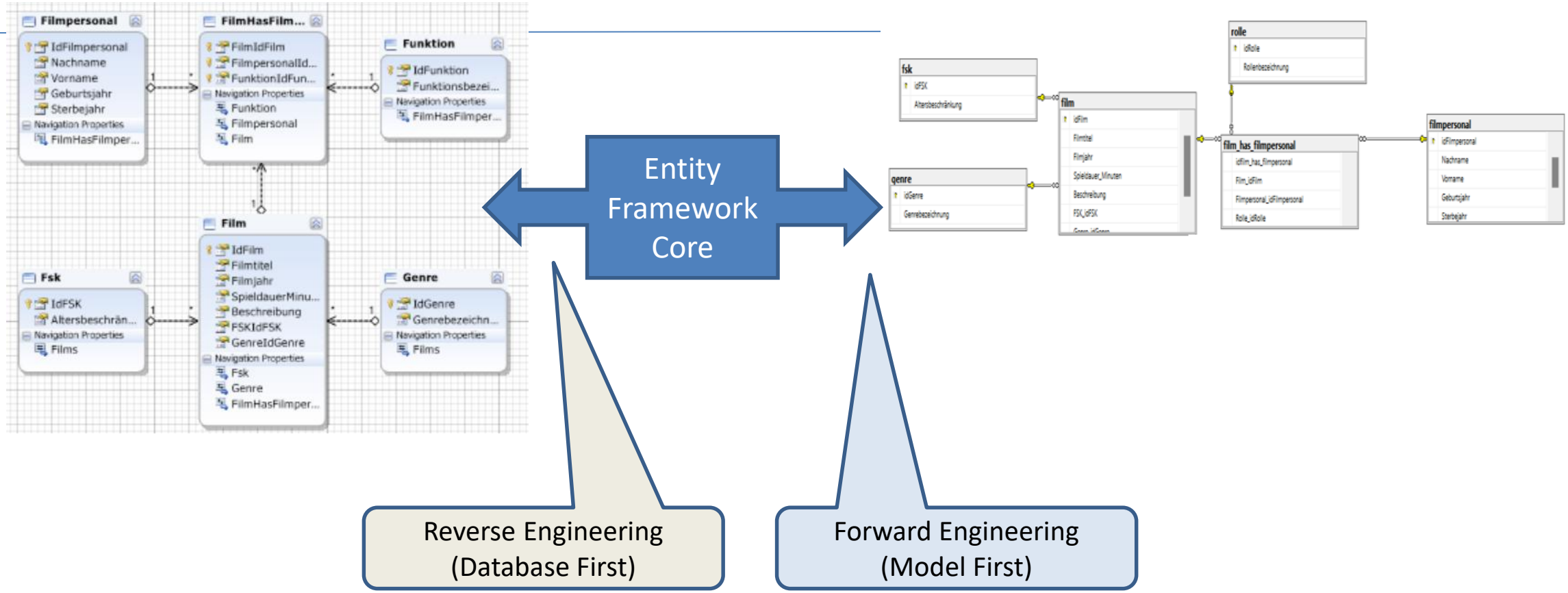
```
using (var dbctx = new EFCDBContext())
{
    var mitProd = dbctx.MitarbeiterListe
          .Where(m => m.Abteilung.Bezeichnung == "Produktion" && m.DatumAustritt ==null);
    Display($"Mitarbeiter der Abt. Produktion:");
    //Display(mitProd.ToQueryString());//Ausgabe des SQL-Statements
    foreach (var m in mitProd.ToList())
    {
          Display($"{m.Nachname}, {m.Vorname}, Mitarbeiter seit:
          {m.DatumEintritt.Year}");
     }
}
```

Migrationen

- Eine Migration besteht aus mehreren Dateien und beschreibt, wie die Entity-Klassen (POCO) und Beziehungen, definiert im DbContext, in die Datenbank übersetzt werden. Diese sind dementsprechend auch **providerspezifisch**!

- Wir unterscheiden
  - Forward Engineering (Model First)
  - Reverse Engineering (Database First)

Das Grundkonzept des Objekt-Relationalen-Mappings (ORM) -Migration



Entity Framework Core unterstützt sowohl das

- Reverse Engineering (alias Database First) bestehender Datenbanken
  → hier werden Klassen aus einem bestehenden Datenbankschema erzeugt

als auch das

- Forward Engineering (alias Code First) einer Datenbank aus Klassen heraus
  → hier wird ein Datenbankschema aus einem Klassenmodell erzeugt).