عنوان: **روش های ارث بری در Entity Framework - قسمت اول**
نویسنده: بهروز راد
تاریخ: ۱۳۹۲/۰۲/۱۷ ۸:۴۰
آدرس: www.dotnettips.info
گروه‌ها: Entity framework, Inheritance Strategies, Table per type, TPT

بخش هایی از کتاب "مرجع کامل Entity Framework 6.0"
ترجمه و تالیف: بهروز راد
وضعیت: در حال نگارش

پیشتر، آقای نصیری در بخشی از مباحث مربوط به Code First در مورد روش‌های مختلف ارث بری در EF و در روش Code First صحبت کرده اند. در این مقاله‌ی دو قسمتی، در مورد دو تا از این روش‌ها در حالت Database First می‌خوانید.

چرا باید از ارث بری استفاده کنیم؟

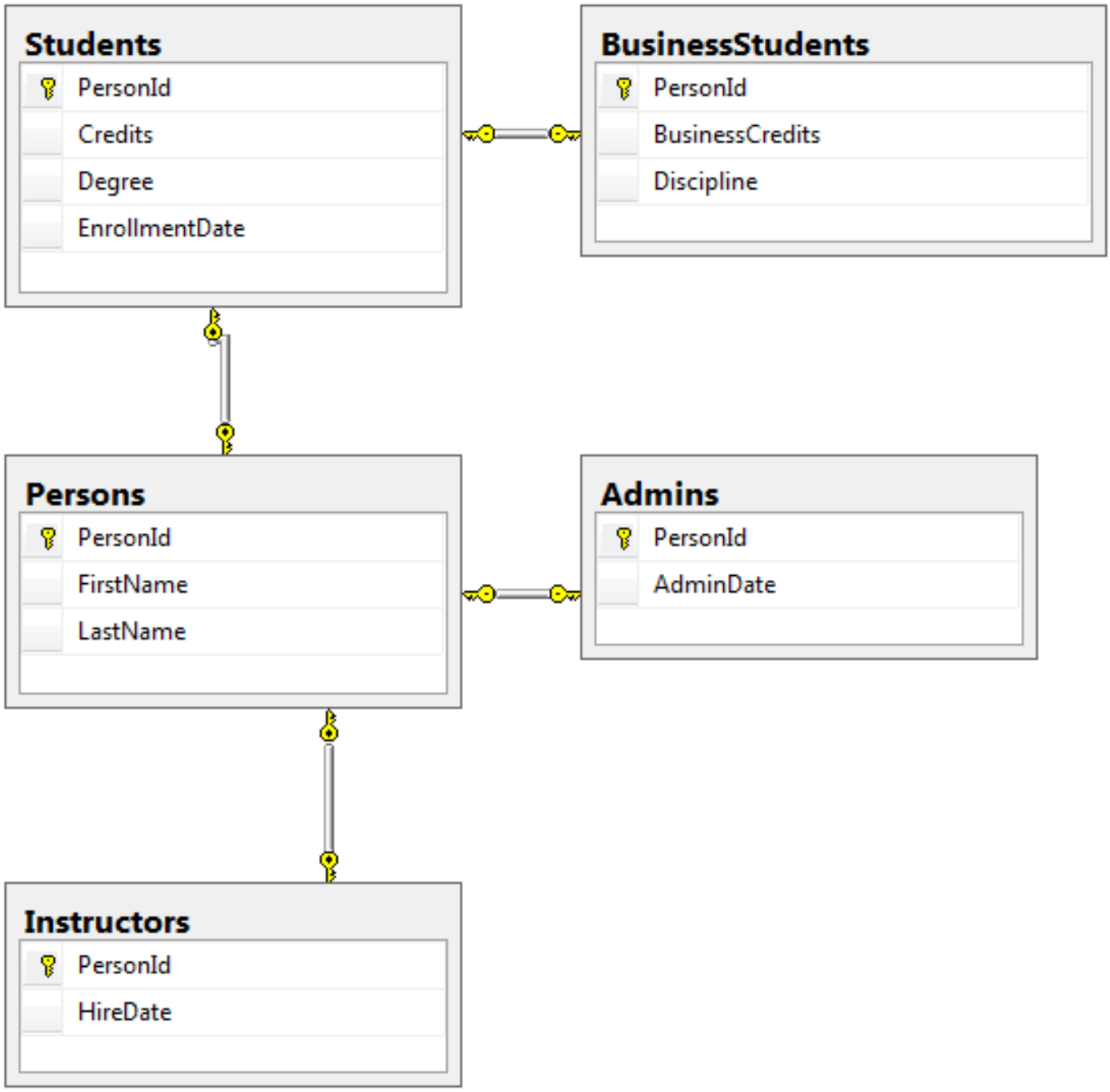
یکی از اهداف اصلی ORMها این است که با ایجاد یک مدل مفهومی از پایگاه داده، آن را هر چه بیشتر به طرز تفکر ما از مدل شی گرای برنامه مان نزدیکتر کنند. از آنجا که ما توسعه گران از مفاهیم شی گرایی مانند "ارث بری" در کدهای خود استفاده می‌کنیم، نیاز داریم تا این مفهوم را در سطح پایگاه داده نیز داشته باشیم. آیا این کار امکان پذیر است؟ EF چه امکاناتی برای رسیدن به این هدف برای ما فراهم کرده است؟ در این قسمت به این سوال پاسخ خواهیم داد.

ارث- بری جداول مفهومی است که در EF به راحتی قابل پیاده- سازی است. سه روش برای پیاده- سازی این مفهوم در مدل وجود دارد.

Table Per Type یا TPT: خصیصه‌های مشترک در جدول پایه قرار دارند و به ازای هر زیر مجموعه نیز یک جدول جدا ایجاد می‌شود.
Table Per Hierarchy یا TPH: تمامی خصیصه‌ها در یک جدول وجود دارند.
Table Per Concrete Type یا TPC: جدول پایه ای وجود ندارد و به ازای هر موجودیت دقیقاً یک جدول همراه با خصیصه‌های موجودیت در آن ایجاد می‌شود.

روش TPT

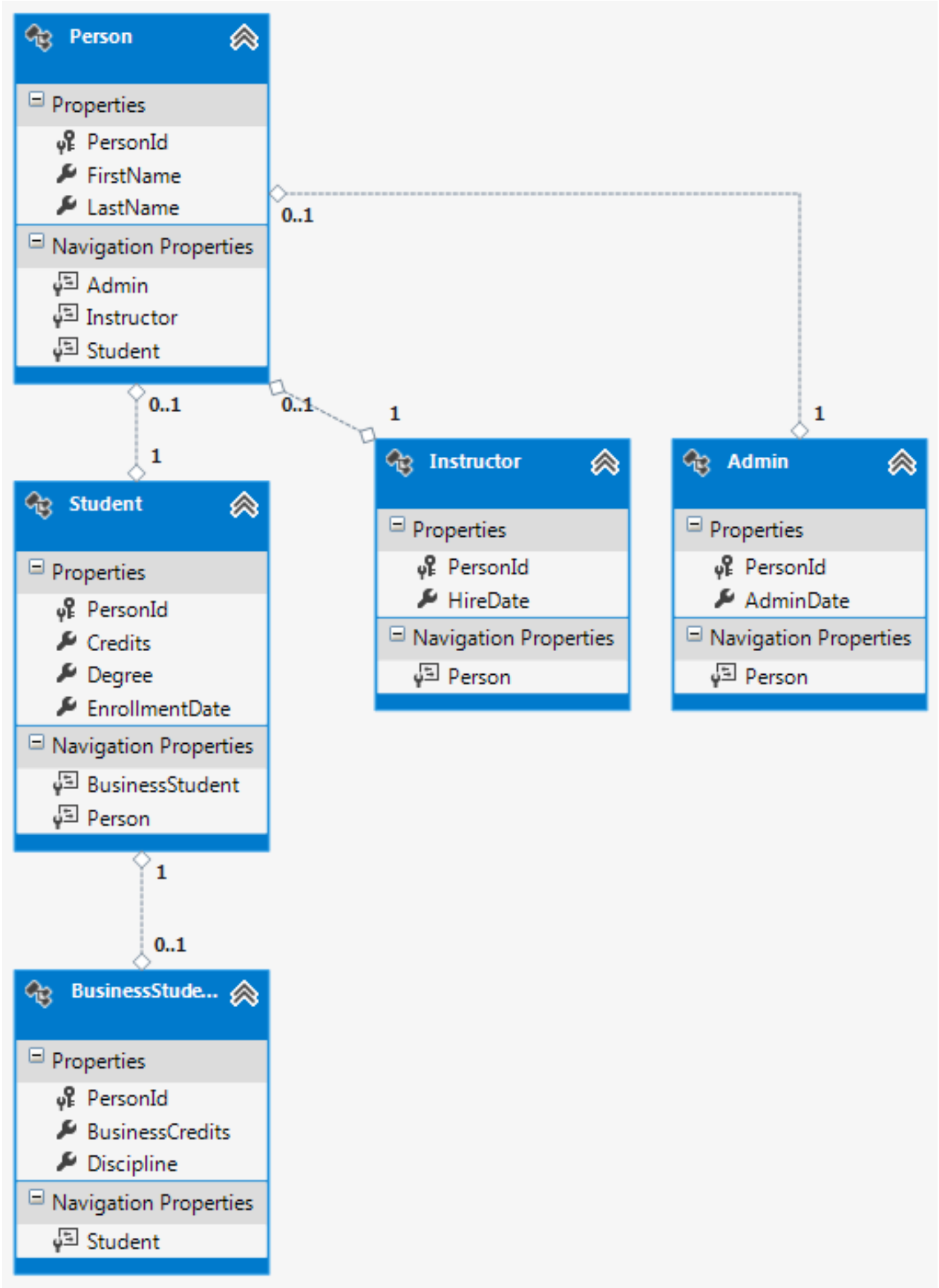
در این روش، خصیصه‌های مشترک در یک جدول پایه قرار دارند و به ازای هر زیر مجموعه از جدول پایه، یک جدول با خصیصه‌های منحصر به آن نوع ایجاد می‌شود. ابتدا جداول و ارتباطات بین آنها که در توضیح مثال برای این روش با آنها کار می‌کنیم را ببینیم.



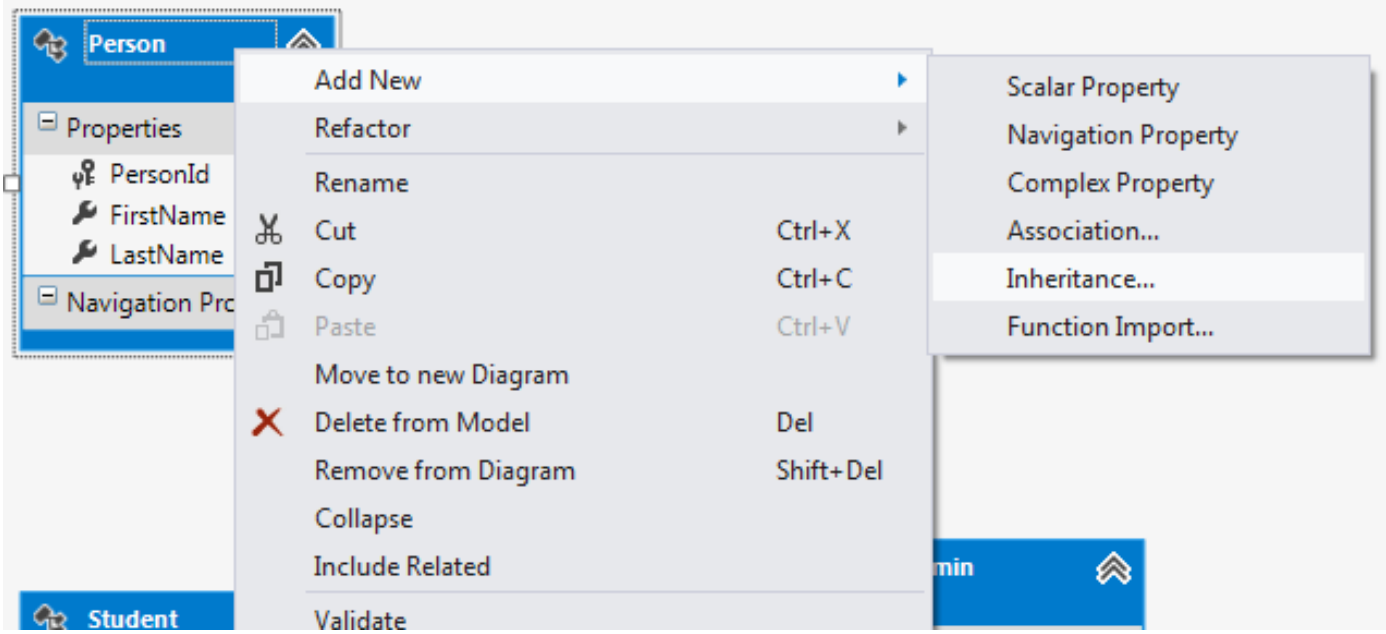
فرض کنید قصد داریم تا در هنگام ثبت مشخصات یک دانش آموز، مقطع تحصیلی او نیز حتماً ذخیره شود. در این حالت، فیلدی با نام Degree ایجاد و تیک گزینه‌ی Allow Nulls را از روبروی آن بر میداریم. با این حال اگر مشخصات دانش آموزان را در جدولی عمومی مثلاً با نام People ذخیره کنیم و این جدول را مکانی برای ذخیره‌ی مشخصات افراد دیگری مانند مدیران و معلمان نیز در نظر بگیریم، از آنجا که قصد ثبت مقطع تحصیلی برای مدیران و معلمان را نداریم، وجود فیلد Degree در کار ما اختلال ایجاد می‌کند. اما با ذخیره‌ی اطلاعات مدیران و معلمان در جداول مختص به خود، می‌توان قانون غیر قابل Null بودن فیلد Degree برای دانش آموزان را به راحتی پیاده سازی کرد.

همان طور که در شکل قبل نیز مشخص است، ما یک جدول پایه با نام Persons ایجاد کرده ایم و خصیصه‌های مشترک بین زیر مجموعه‌ها (FirstName و LastName) را در آن قرار داده ایم. سه موجودیت (Student، Admin و Instructor) از Persons ارث می‌برند و موجودیت BusinessStudent نیز از Student ارث می‌بَرَد.

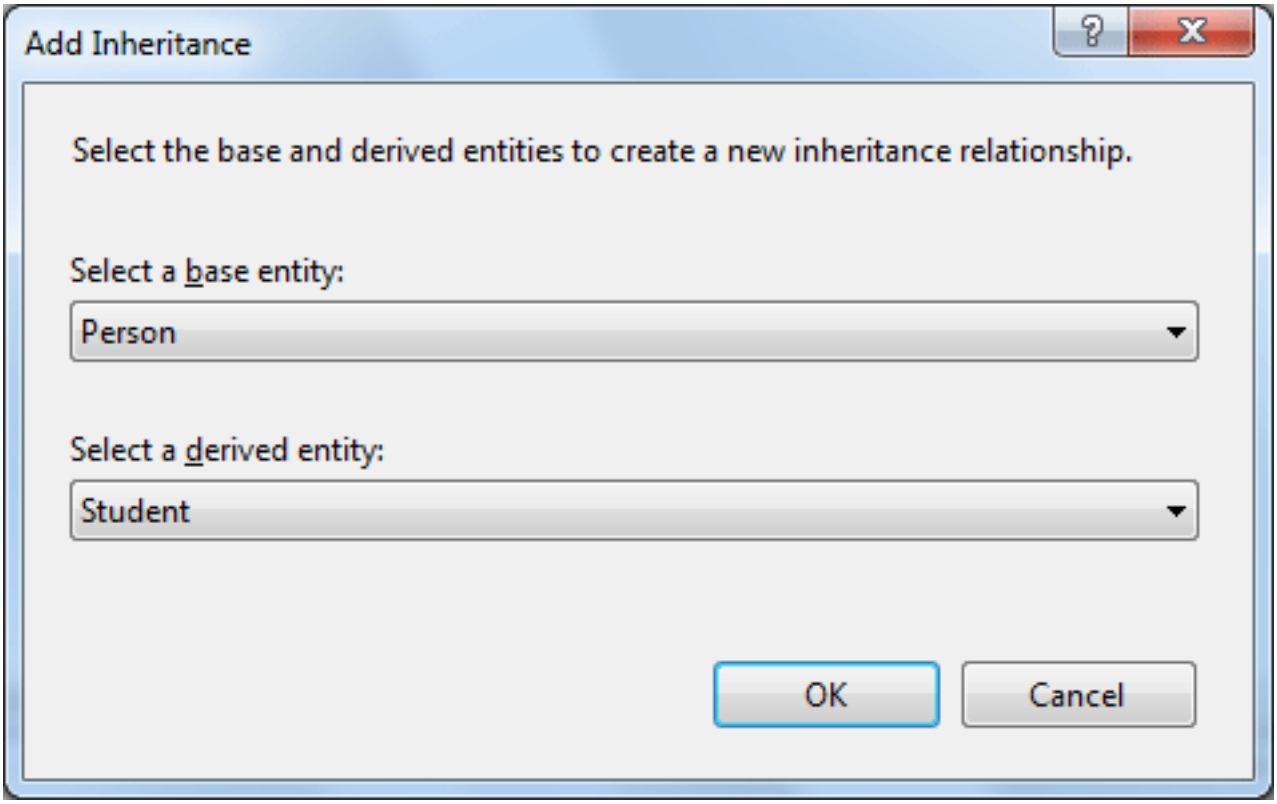
جداول ایجاد شده، پس از ایجاد مدل به روش Database First، به شکل زیر تبدیل می‌شوند.


Person
Properties
PersonId
FirstName
LastName
Navigation Properties
Admin
Instructor
Student
0..1
0..1
1
1
0..1
1
Student
Properties
PersonId
Credits
Degree
EnrollmentDate
Navigation Properties
BusinessStudent
Person
Instructor
Properties
PersonId
HireDate
Navigation Properties
Person
Admin
Properties
PersonId
AdminDate
Navigation Properties
Person
1
0..1
BusinessStude...
Properties
PersonId
BusinessCredits
Discipline
Navigation Properties
Student

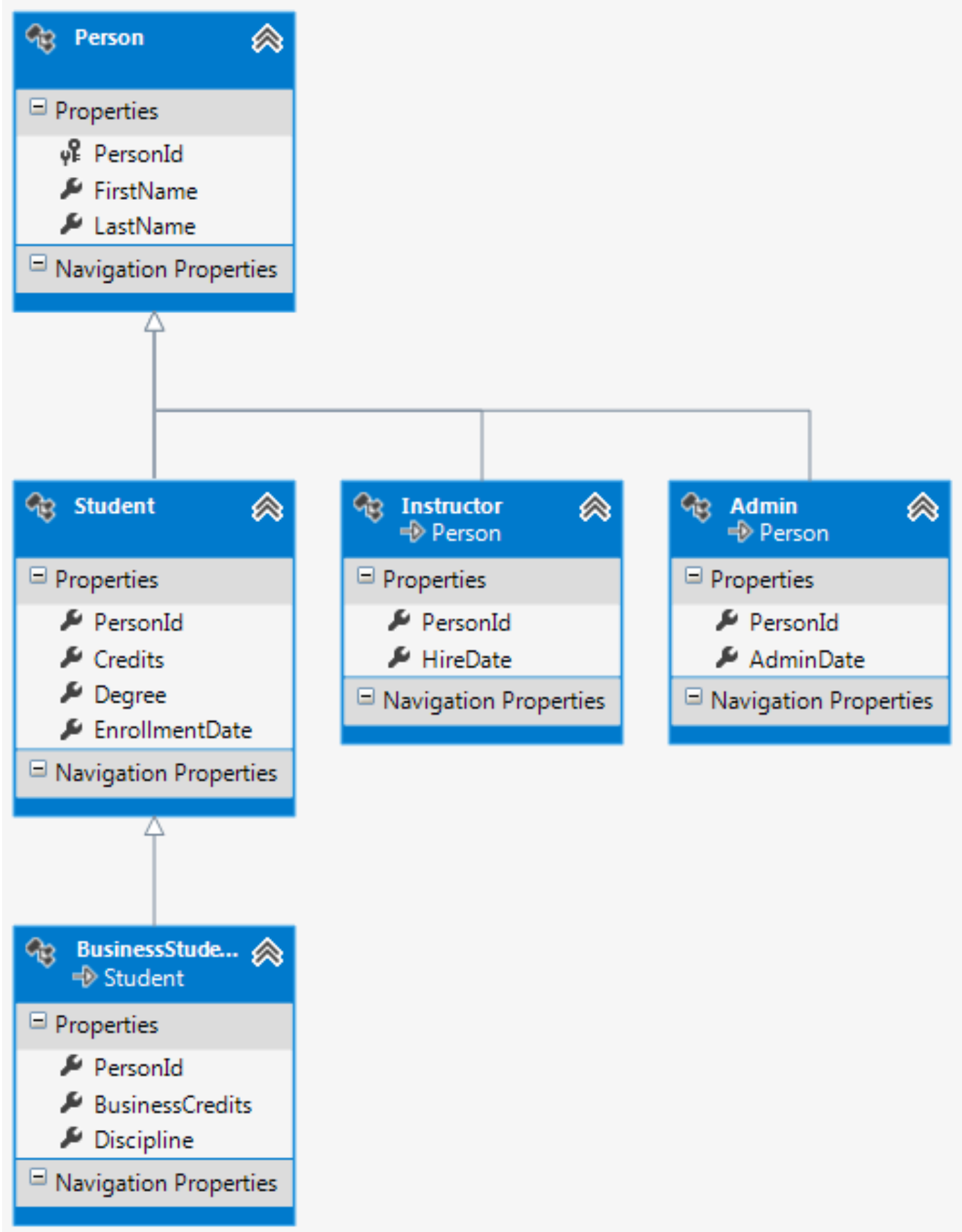
از آنجا که قصد داریم ارتباطات ارث بری شده ایجاد کنیم، باید ارتباطات پیش فرض شکل گرفته بین موجودیت‌ها را حذف کنیم. بدین منظور، بر روی هر خط ارتباطی در EDM Designer کلیک راست و گزینه‌ی Delete from Model را انتخاب کنید. سپس بر روی موجودیت Person، کلیک راست کرده و از منوی Add New، گزینه‌ی Inheritance را انتخاب کنید (شکل زیر).



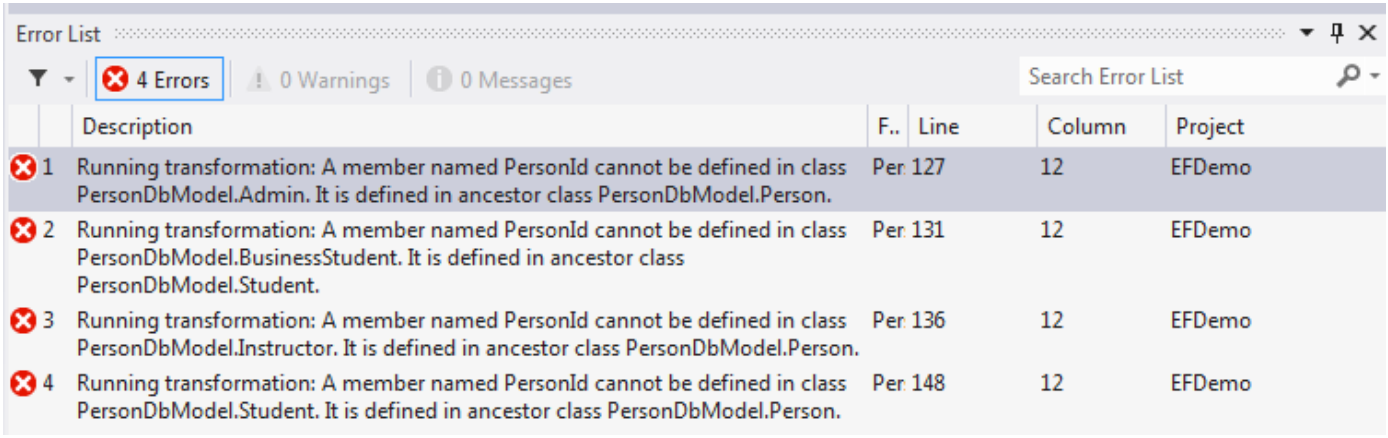
شکل زیر ظاهر می‌شود.



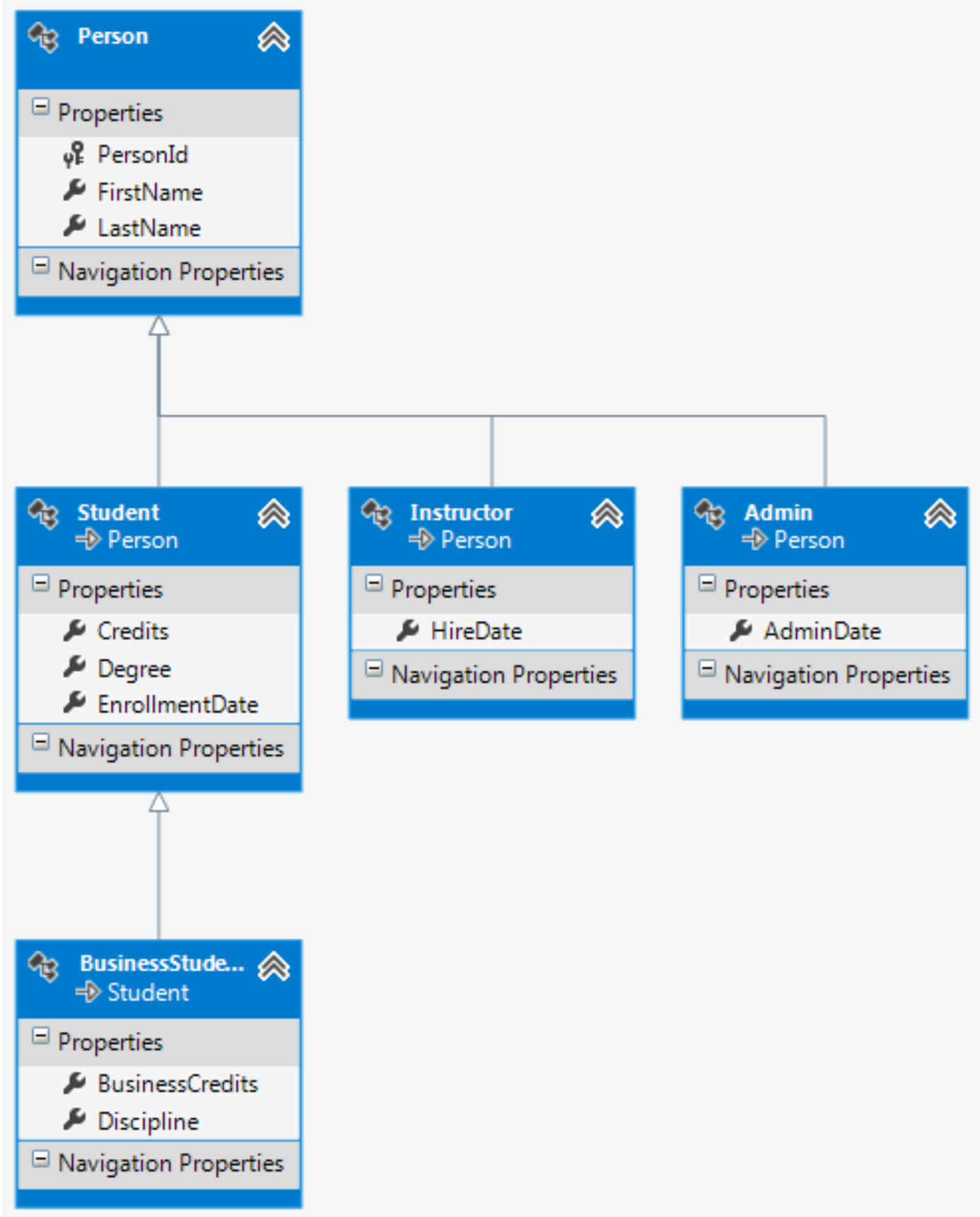
قسمت بالا، موجودیت پایه، و قسمت پایین، موجودیت مشتق شده را مشخص می‌کند. این کار را سه مرتبه برای ایجاد ارتباط ارث بری شده بین موجودیت Person به عنوان موجودیت پایه و موجودیت‌های Instructor، Student و Admin به عنوان موجودیت‌های مشتق شده ایجاد کنید. همچنین یک ارتباط نیز بین موجودیت Student به عنوان موجودیت پایه و موجودیت BusinessStudent به عنوان موجودیت مشتق شده ایجاد کنید. نتیجه‌ی کار را در شکل زیر ملاحظه می‌کنید.



اگر بر روی دکمه‌ی Save در نوار ابزار Visual Studio کلیک کنید، چهار خطا در پنجره‌ی Error List نمایش داده می‌شود



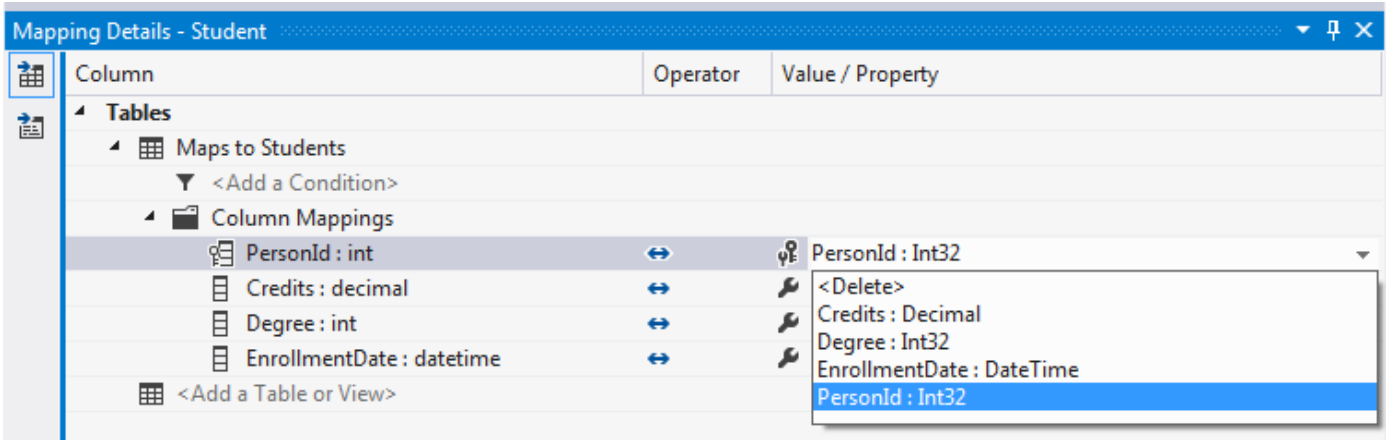
این خطاها بیانگر این هستند که خصیصه‌ی PersonId به دلیل اینکه در موجودیت پایه‌ی Person تعریف شده است، نباید در موجودیت‌های مشتق شده از آن نیز وجود داشته باشد چون موجودیت‌های مشتق شده، خصیصه‌ی PersonId را به ارث برده اند. وجود این خصیصه در زمان طراحی جدول در مدل فیزیکی الزامی بوده است اما اکنون ما با مدل مفهومی و قوانین شی گرایی سر و کار داریم. بنابراین خصیصه‌ی PersonId را از موجودیت‌های Admin، Instructor، Student و BusinessStudent حذف کنید.
شکل زیر، نتیجه‌ی کار را نشان می‌دهد.



اکنون اگر بر روی دکمه‌ی Save کلیک کنید، خطاها از بین می‌روند.

ما خصیصه‌ی PersonId را از موجودیت‌های مشتق شده به این دلیل که آن را از موجودیت پایه ارث می‌برند حذف کردیم. حال این خصیصه برای موجودیت‌های مشتق شده وجود دارد اما باید مشخص کنیم که به کدام خصیصه از کلاس پایه تناظر دارد. شاید

انتظار این باشد که EF، خود تشخیص بدهد که PersonId در موجودیت‌های مشتق شده باید به PersonId کلاس پایه‌ی خود تناظر داشته باشد اما در حال حاضر این کاری است که خود باید انجام دهیم. بدین منظور، بر روی هر یک از موجودیت‌های مشتق شده کلیک راست کرده و گزینه‌ی Table Mapping را انتخاب کنید. سپس همان طور که در شکل زیر مشاهده می‌کنید، تناظر را ایجاد کنید.



مدل ما آماده است. آن را امتحان می‌کنیم. در زیر، یک کوئری LINQ ساده بر روی مدل ایجاد شده را ملاحظه می‌کنید.

```
using (PersonDbEntities context = new PersonDbEntities())
{

    var people = from p in context.Persons
                 select p;

    foreach (Person person in people)
    {
        Console.WriteLine("{0}, {1}",
            person.LastName,
            person.FirstName);
    }

    Console.ReadLine();
}
```

قضیه به همین جا ختم نمی‌شود! ما الان یک مدل ارث بری شده داریم. بهتر است مزایای آن را در عمل ببینیم. شاید دوست داشته باشیم تا فقط اطلاعات زیر مجموعه‌ی BusinessStudent را بازیابی کنیم.

```
using (PersonDbEntities context = new PersonDbEntities())
{

    var students = from p in context.Persons.OfType<BusinessStudent>()
                   select p;

    foreach (BusinessStudent student in students)
    {
        Console.WriteLine("{0}, {1}: Degree {2}, Discipline {3}",
            student.LastName,
            student.FirstName,
            student.Degree,
            student.Discipline);
    }

    Console.ReadLine();
}
```

همان طور که در کدهای قبل نیز مشخص است، خصیصه‌های LastName و FirstName از موجودیت پایه یعنی Person، خصیصه‌ی Degree از موجودیت مشتق شده‌ی Student (که البته در نقش موجودیت پایه برای BusinessStudent است) و Discipline از موجودیت مشتق شده یعنی BusinessStudent خوانده می‌شوند.
یک روش دیگر نیز برای کار با این سلسه مراتب ارث بری وجود دارد. کوئری اول را دست نزنیم (اطلاعات موجودیت پایه را بازیابی کنیم) و پیش از انجام عملیاتی خاص، نوع موجودیت مشتق شده را بررسی کنیم. مثالی در این زمینه:

```
using (PersonDbEntities context = new PersonDbEntities())
{

    var people = from p in context.Persons
                 select p;

    foreach (Person person in people)
    {
        Console.WriteLine("{0}, {1}",
            person.LastName,
            person.FirstName);

        if (person is Student)
            Console.WriteLine("    Degree: {0}",
                ((Student)person).Degree);

        if (person is BusinessStudent)
            Console.WriteLine("    Discipline: {0}",
                ((BusinessStudent)person).Discipline);
    }

    Console.ReadLine();
}
```

مزایای روش TPT

امکان نرمال سازی سطح 3 در این روش به خوبی وجود دارد
افزونگی در جداول وجود ندارد.
اصلاح مدل آسان است (برای اضافه یا حذف کردن یک موجودیت به/از مدل فقط کافی است تا جدول متناظر با آن را از پایگاه داده حذف کنید)

معایب روش TPT

سرعت عملیات CRUD (ایجاد، بازیابی، آپدیت، حذف) داده‌ها با افزایش تعداد موجودیت‌های شرکت کننده در سلسله مراتب ارث بری کاهش می‌یابد. به عنوان مثال، کوئری‌های SELECT، حاوی عبارت‌های JOIN خواهند بود و عدم توجه صحیح به کوئری نوشته شده می‌تواند منجر به حضور چندین عبارت JOIN که برای ارتباط بین جداول به کار می‌رود در اسکریپت تولیدی و کاهش زمان اجرای بازیابی داده‌ها شود.
تعداد جداول در پایگاه داده زیاد می‌شود

در قسمت بعد با روش TPH آشنا می‌شوید.

نظرات خوانندگان

نویسنده: سید امیر سجادی
تاریخ: ۱۳۹۲/۰۲/۱۷ ۲۳:۲۷

با سلام
به نظر شما برای پروژه‌های بزرگ اگه از LINQ TO SQL استفاده بشه کارایی و سرعت رو پایین می‌آره یا نه؟

نویسنده: مهمان
تاریخ: ۱۳۹۲/۰۲/۱۹ ۱۲:۶

سلام آقای راد

ضمن تشکر بابت مطالبی که نوشتید راستی کتاب مرجع کامل EF 6 کی به بازار میاد؟

نویسنده: بهروز راد
تاریخ: ۱۳۹۲/۰۲/۲۰ ۹:۲۴

از LINQ To Entities و Entity Framework استفاده کنید. بهینه‌تر هست.

نویسنده: بهروز راد
تاریخ: ۱۳۹۲/۰۲/۲۰ ۹:۲۵

حدوداً یک ماه پس از انتشار نسخه‌ی نهایی EF 6.0 توسط مایکروسافت.

نویسنده: کوروش شاهی
تاریخ: ۱۳۹۳/۰۲/۲۴ ۱۷:۱۴

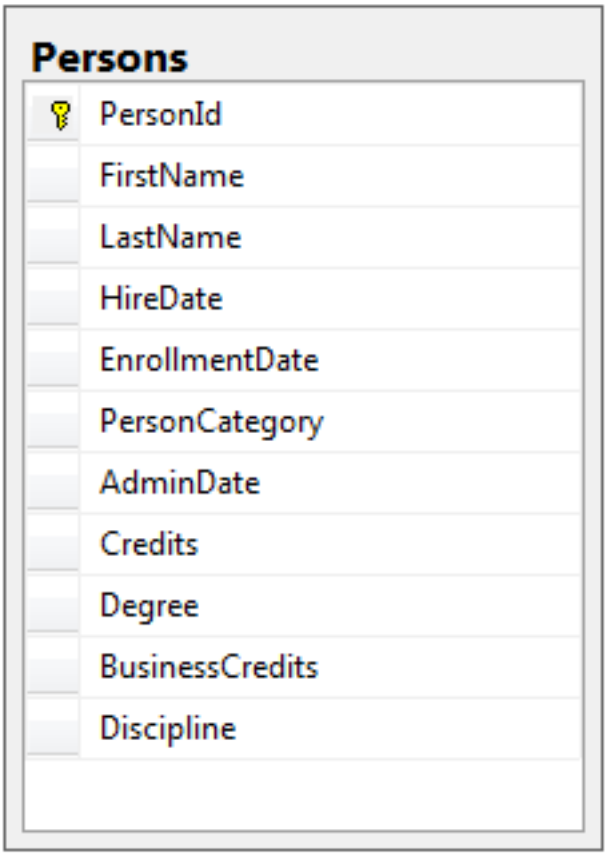
سلام
هنوز کتاب انتشار نیافته است ؟

عنوان: **روش های ارث بری در Entity Framework - قسمت دوم**
نویسنده: بهروز راد
تاریخ: ۱۳۹۲/۰۲/۲۱ ۸:۴۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework, Inheritance Strategies, Table per hierarchy, TPH

در قسمت اول در مورد روش TPT خواندید. در این قسمت به روش TPH می‌پردازیم.

روش TPH

در این روش، ارث بری از طریق فقط یک جدول ایجاد می‌شود و زیر مجموعه‌ها بر اساس مقدار یک فیلد از یکدیگر متمایز می‌شوند. پس اگر جدولی دارید که برای متمایز کردن رکوردهای آن از یک فیلد استفاده می‌کنید، روش TPH مناسب شما است. با روش TPH نیز می‌توانید به همان مدلی که در روش TPT دارید برسید، تنها تفاوت این هست که در روش TPH، تمامی داده‌ها در یک جدول قرار دارند و یک فیلد برای متمایز کردن رکوردها استفاده می‌شود. همه چیز با مثال عملی واضح‌تر است. پس کار خود را با یک مثال ادامه می‌دهیم. جدول مثال ما در شکل زیر مشخص است.



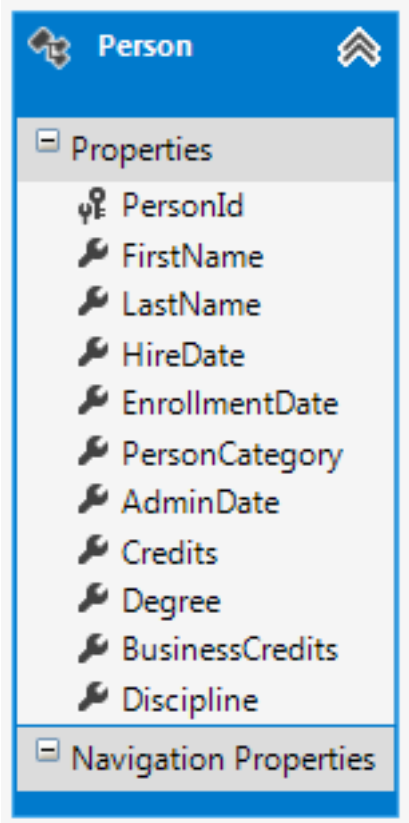
به نظر می‌رسد که این جدول با جداول قسمت قبل شباهتی دارد. بله! فیلدهای جداول مثال قبل در این جدول آمده اند.
فیلدهای FirstName و LastName از جدول Persons
فیلد HireDate از جدول Instructors
فیلد EnrollmentDate، Credits و Degree از جدول Students
فیلد AdminDate از جدول Admins
فیلدهای BusinessCredits و Discipline از جدول BusinessStudents

یک فیلد با نام PersonCategory نیز اضافه شده است که «مقداری عددی» می‌پذیرد و برای «متمایز کردن رکوردها» استفاده

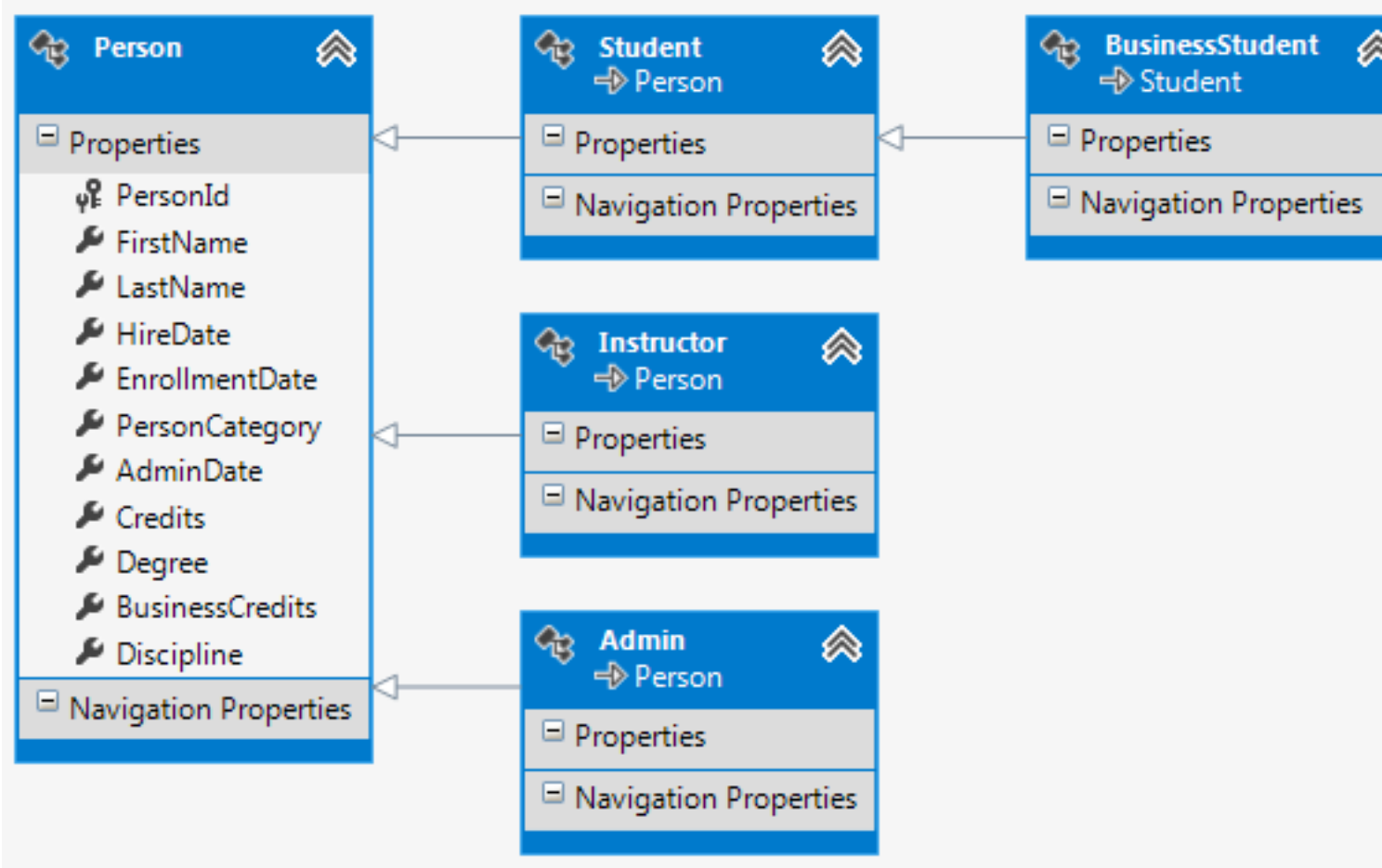
می‌شود:

1 ، نمایانگر Student
2 ، نمایانگر Instructor
3 ، نمایانگر Admin
4 ، نمایانگر BusinessStudent

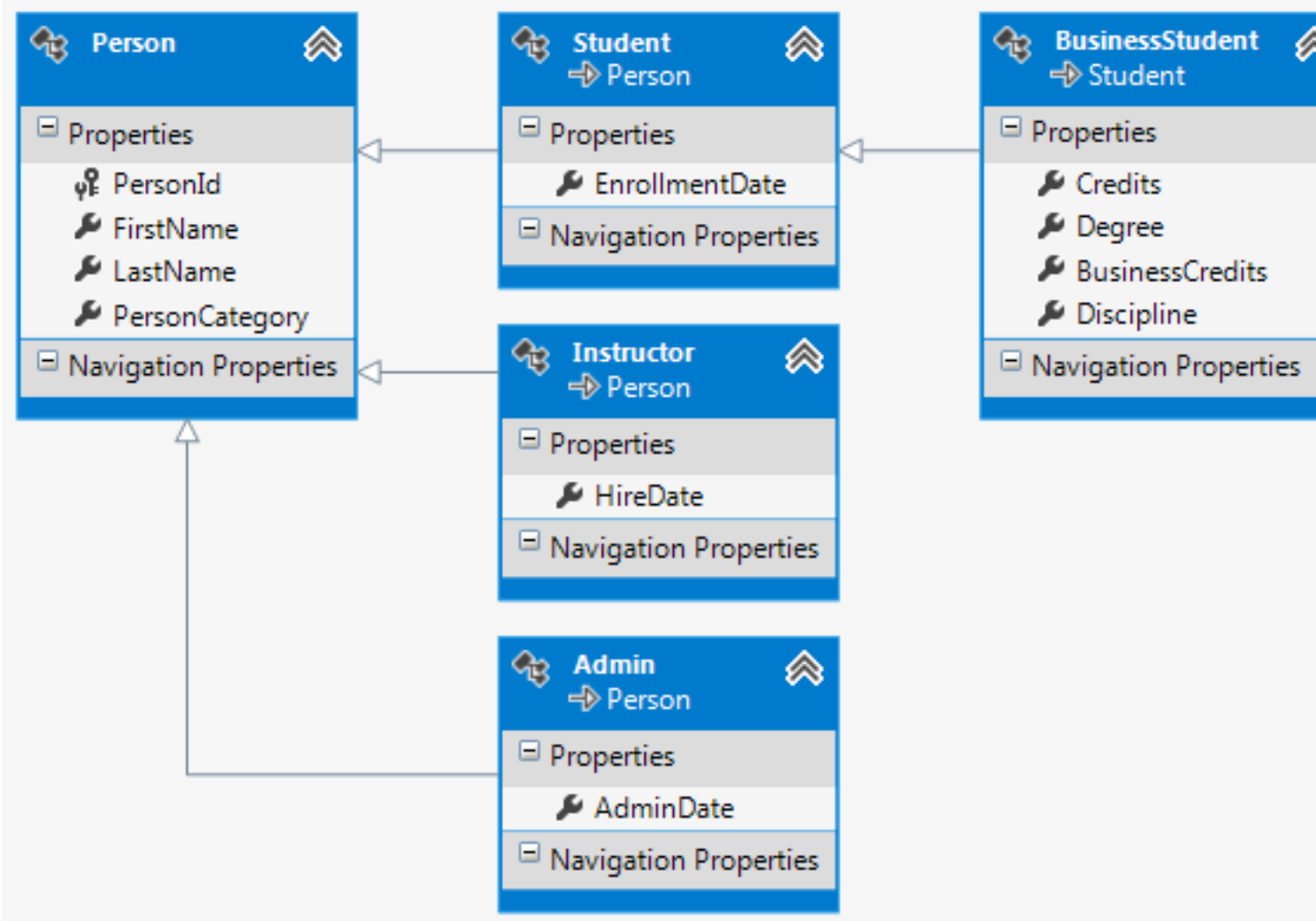
از این جدول می‌خواهیم به مدل قسمت اول برسیم اما این بار با استفاده از روش TPH. در شکل زیر، جدول Persons به صورت مدل شده در برنامه نشان داده شده است.



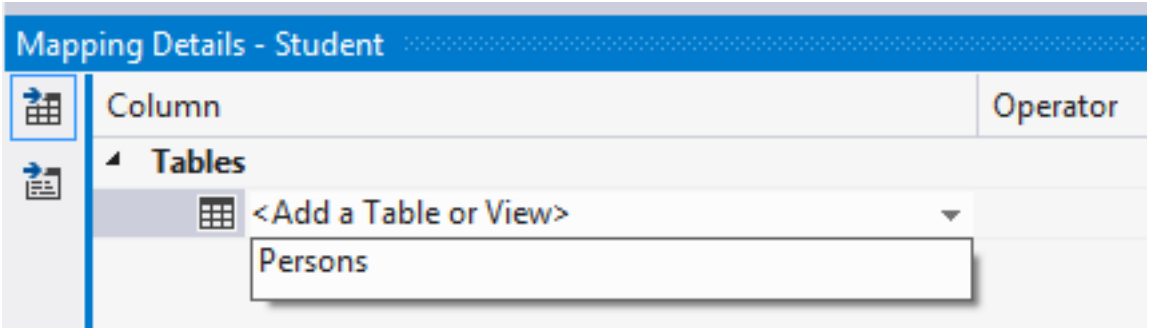
حال باید چهار موجودیت مشتق شده را به مدل اضافه کنیم. موجودیت‌های Student، Instructor و Admin را با کلیک راست بر روی EDM Designer و انتخاب گزینه‌ی Entity از منوی Add New ایجاد کنید. در فرمی که باز می‌شود، از قسمت Base type، Person را انتخاب کنید. موجودیت BusinessStudent را نیز ایجاد و موجودیت پایه‌ی آن را موجودیت Student در نظر بگیرید. مدل ایجاد شده تا اینجای کار در شکل زیر مشخص است.



حال باید خصیصه‌های موجودیت Person را به موجودیت‌های مشتق شده منتقل کرد. بدین منظور، هر خصیصه از موجودیت Person را انتخاب، کلیدهای Ctrl+X را فشار دهید، سپس بر روی قسمت Properties موجودیت مشتق شده‌ی مورد نظر رفته و کلیدهای Ctrl+V را فشار دهید. نتیجه در شکل زیر نشان داده شده است.

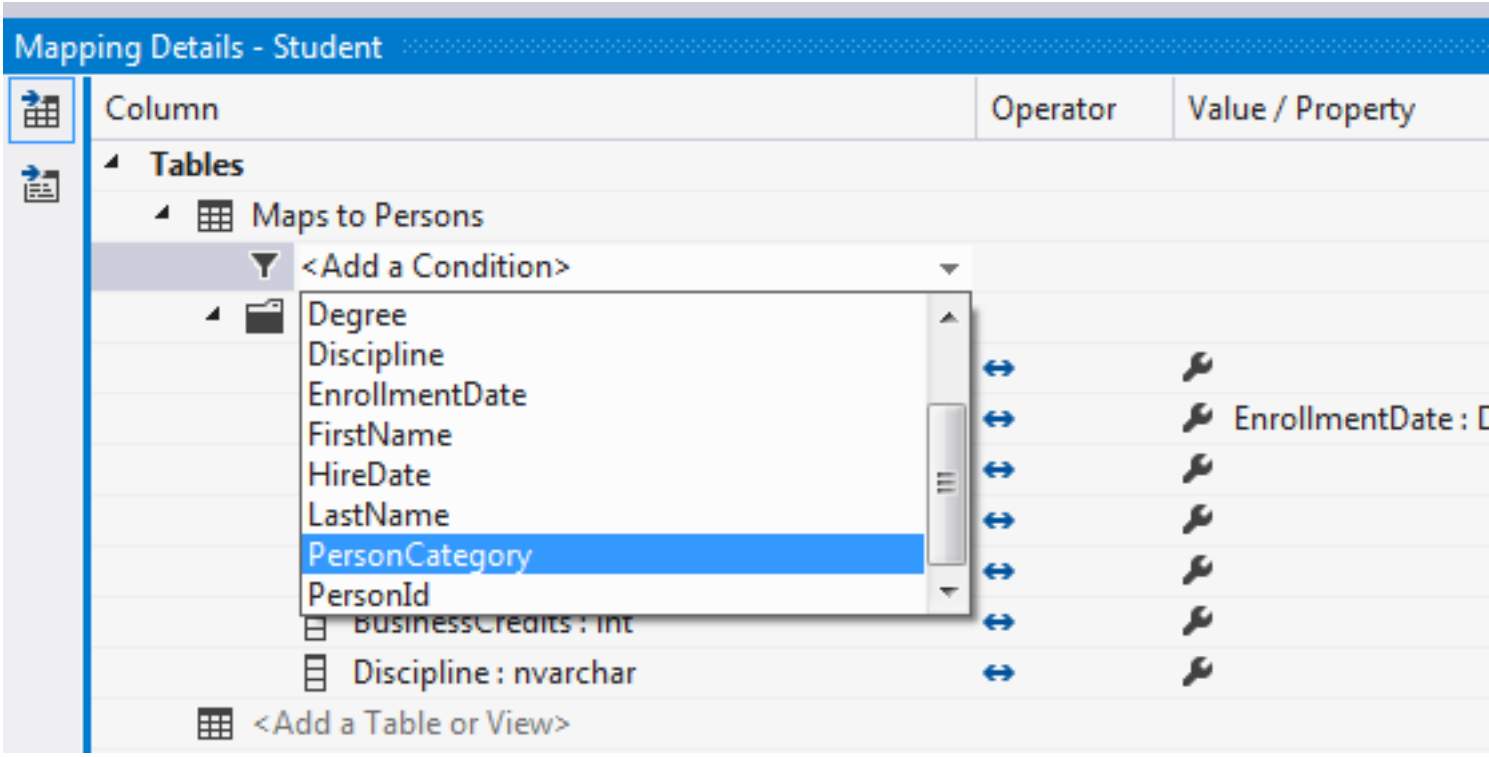


اکنون زمان آن رسیده است تا جدول متناظر با هر یک از موجودیت‌های مشتق شده را معرفی کنیم. تمامی موجودیت‌های مشتق شده از جدول Persons استفاده می‌کنند. بر روی هر یک از آنها کلیک راست کرده و گزینه‌ی Table Mapping را انتخاب کنید.
پنجره ی Mapping Details نشان داده می‌شود. ابتدا بر روی عبارت Add a Table or View و سپس بر روی نشانگر رو به پایینی که کنار آن ظاهر می‌شود کلیک کنید (شکل زیر).

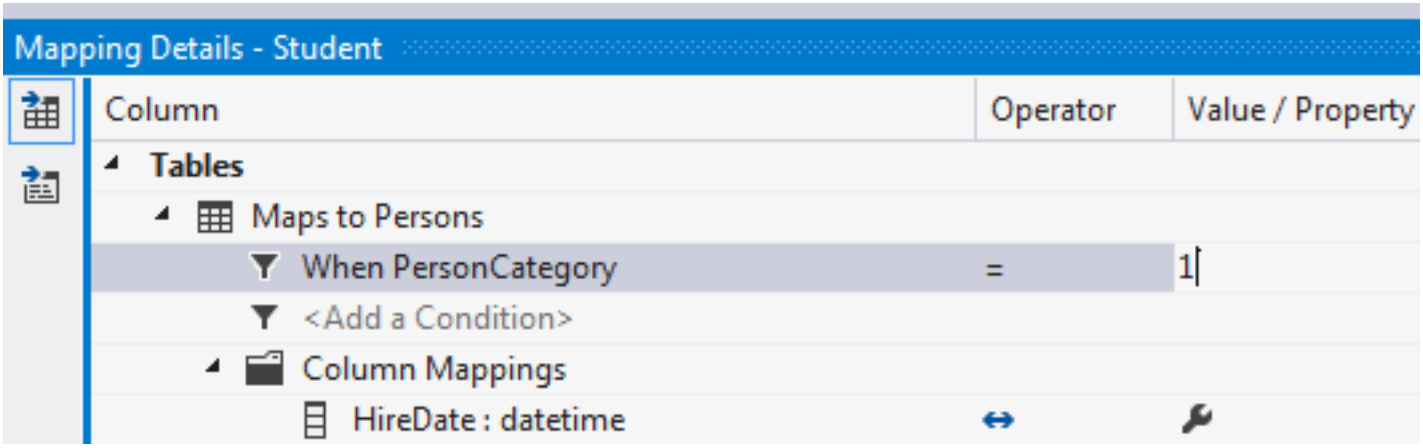


آیتم Persons را انتخاب کنید. اکنون باید فیلد تفکیک کننده‌ی رکوردها را مشخص کنیم. برای این حالت باید یک شرط ایجاد نمود. در همان پنجره‌ی Mapping Details، عبارتی با عنوان Add a Condition وجود دارد. بر روی آن کلیک و در لیستی که ظاهر می‌شود،

آیتم PersonCategory را انتخاب کنید (شکل زیر).



سپس در ستون Value/Property، مقدار آن را "1" قرار دهید (شکل زیر).



تناظر میان موجودیت و جدول Persons و مقداردهی مناسب به فیلد متمایز کننده را برای تمامی موجودیت‌های مشتق شده انجام دهید. دلیل این کار این است که EF بداند هر رکورد در چه زمانی باید به چه موجودیتی تبدیل شود. دقت کنید که پیشتر به مقدار فیلد متمایز کننده برای هر موجودیت اشاره کردیم. نکته‌ی مهم اینکه یک شرط نیز باید برای موجودیت Person ایجاد و مقدار فیلد متمایز کننده‌ی آن را "صفر" تعریف کنید.
مثال ما آماده است. آن را امتحان می‌کنیم.

```
using (PersonDbEntities context = new PersonDbEntities())
{

    var people = from p in context.Persons
                 select p;

    foreach (Person person in people)
    {
        Console.WriteLine("{0}, {1}",
            person.LastName,
            person.FirstName);

        if (person is Student)
            Console.WriteLine("    Degree: {0}",
                ((Student)person).Degree);

        if (person is BusinessStudent)
            Console.WriteLine("    Discipline: {0}",
                ((BusinessStudent)person).Discipline);
    }

    Console.ReadLine();
}
```

چه کدهای آشنایی! بله، این کدها همان کدهایی هستند که برای مثال روش TPT استفاده کردیم و بدون کوچکترین تغییری در اینجا نیز قابل استفاده هستند.

مزایای روش TPH

سرعت بالای عملیات CRUD، به دلیل وجود تمامی داده‌ها در یک جدول

تعداد جداول در پایگاه داده، کم و مدیریت آنها آسان‌تر است

معایب روش TPH

افزونگی داده ها. مقادیر برخی ستون‌ها برای بعضی از رکوردها، حاوی مقدار NULL است و تعداد این ستون‌ها به تعداد زیر مجموعه‌ها ارتباط دارد

عیب اول، باعث می‌شود تا در صورتی که داده‌ها به صورت دستی تغییر پیدا کنند، جامعیت داده‌ها از بین برود

افزایش بی دلیل حجم داده ها

اضافه و حذف کردن موجودیت‌ها به مدل، عملی زمانبر و پیچیده است

سومین روش، TPC است که در EF Designer، پشتیبانی از پیش تعبیه شده برای آن وجود ندارد و باید از طریق ویرایش فایل edmx. به آن رسید. استفاده از این روش توصیه نمی‌شود.

نظرات خوانندگان

نویسنده: محمد فریدونی
تاریخ: ۱۳۹۲/۰۵/۱۹ ۱۳:۲۰

سلام؛ با توجه به دو مدلی که در ارث بری می‌تونیم بکار ببریم ، روش مناسب و بهتر، با توجه مزایا و معایبی که فرمودید کدوم روش است ؟

روش TPT ؟

یا روش

روش TPH ؟

نویسنده: محمد پهلوان
تاریخ: ۱۳۹۲/۰۸/۲۱ ۱۳:۴۶

روش‌های ذکر شده در اینجا نیز بررسی شده است

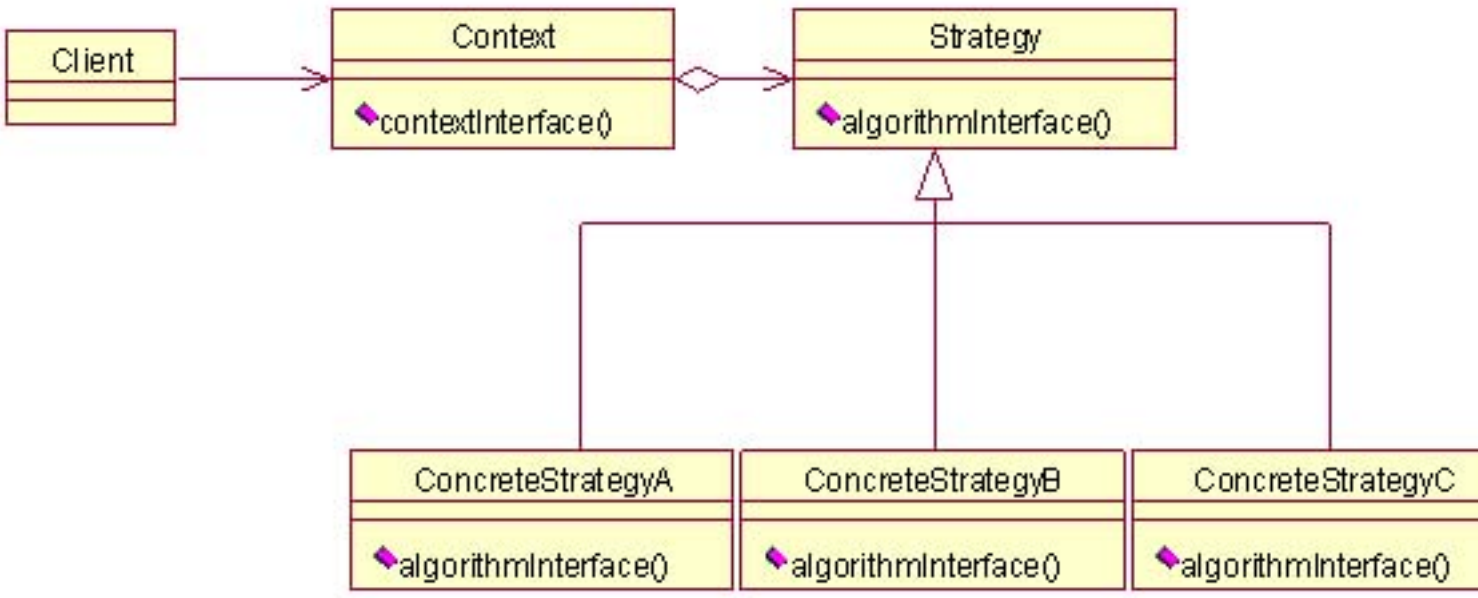
عنوان: **الگوی استراتژی - Strategy Pattern**
نویسنده: مجتبی شاطری
تاریخ: ۱۳۹۲/۰۶/۲۰ ۰:۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, Dependency Injection, SOLID Principals, Inheritance Strategies, SOLID, Dependency Inversion

الگوی استراتژی (Strategy) اجازه می-دهد که یک الگوریتم در یک کلاس بسته بندی شود و در زمان اجرا برای تغییر رفتار یک شیئ تعویض شود.
برای مثال فرض کنید که ما در حال طراحی یک برنامه مسیریابی برای یک شبکه هستیم. همانطوریکه می‌دانیم برای مسیر یابی الگوریتم‌های مختلفی وجود دارد که هر کدام دارای مزایا و معایبی هستند. و با توجه به وضعیت موجود شبکه یا عملی که قرار است انجام پذیرد باید الگوریتمی را که دارای بالاترین کارائی است انتخاب کنیم. همچنین این برنامه باید امکانی را به کاربر بدهد که کارائی الگوریتم‌های مختلف را در یک شبکه فرضی بررسی کنید. حالا طراحی پیشنهادی شما برای این مسئله چیست؟
دوباره فرض کنید که در مثال بالا در بعضی از الگوریتم‌ها نیاز داریم که گره‌های شبکه را بر اساس فاصله‌ی آنها از گره مبداء مرتب کنیم. دوباره برای مرتب سازی الگوریتم‌های مختلف وجود دارد و هر کدام در شرایط خاص، کارائی بهتری نسبت به الگوریتم‌های دیگر دارد. مسئله دقیقا شبیه مسئله بالا است و این مسله می‌توانند دارای طراحی شبیه مسله بالا باشد. پس اگر ما بتوانیم یک طراحی خوب برای این مسئله ارائه دهیم می‌توانیم این طراحی را برای مسائل مشابه به کار ببریم.

هر کدام از ما می‌توانیم نسبت به درک خود از مسئله و سلیقه کاری، طراح‌های مختلفی برای این مسئله ارائه دهیم. اما یک طراحی که می‌تواند یک جواب خوب و عالی باشد، الگوی استراتژی است که توانسته است بارها و بارها به این مسئله پاسخ بدهد.

الگوی استراتژی گزینه مناسبی برای مسائلی است که می‌توانند از چندین الگوریتم مختلف به مقصود خود برسند.

نمودار UML الگوی استراتژی به صورت زیر است :



اجازه بدهید، شیوه کار این الگو را با مثال مربوط به مرتب سازی بررسی کنیم. فرض کنید که ما تصمیم گرفتیم که از سه الگویتم زیر برای مرتب سازی استفاده کنیم.

1 - الگوریتم مرتب سازی Shell Sort 2 - الگوریتم مرتب سازی Quick Sort
3 - الگوریتم مرتب سازی Merge Sort

ما برای مرتب سازی در این برنامه دارای سه استراتژی هستیم. که هر کدام را به عنوان یک کلاس جداگانه در نظر می‌گیریم (همان کلاس‌های ConcreteStrategy). برای اینکه کلاس Client بتواند به سادگی یک از استراتژی‌ها را انتخاب کنید بهتر است که تمام کلاس‌های استراتژی دارای اینترفیس مشترک باشند. برای این کار می‌توانیم یک کلاس abstract تعریف کنیم و ویژگیهای مشترک کلاس‌های استراتژی را در آن قرار دهیم و کلاس‌های استراتژی آنها را به ارث ببرند(همان کلاس Strategy) و پیاده سازی کنند.

در زیل کلاس Abstract که کل کلاس‌های استراتژی از آن ارث می‌برند را مشاهده می‌کنید :

```
abstract class SortStrategy
{
        public abstract void Sort(ArrayList list);
}
```

کلاس مربوط به QuickSort

```
class QuickSort : SortStrategy
{
        public override void Sort(ArrayList list)
        {
          // الگوریتم مربوطه
        }


}
```

کلاس مربوط به ShellSort

```
class ShellSort : SortStrategy
{
        public override void Sort(ArrayList list)
        {
          // الگوریتم مربوطه
        }
}
```

کلاس مربوط به MergeSort

```
class MergeSort : SortStrategy
{
        public override void Sort(ArrayList list)
        {
          // الگوریتم مربوطه
        }
}
```

و در آخر کلاس Context که یکی از استراتژی‌ها را برای مرتب کردن به کار می‌برد :

```
class SortedList
{
        private ArrayList list = new ArrayList();
        private SortStrategy sortstrategy;

        public void SetSortStrategy(SortStrategy sortstrategy)
        {
            this.sortstrategy = sortstrategy;
        }
        public void Add(string name)
```

```
        {
            list.Add(name);
        }
        public void Sort()
        {
            sortstrategy.Sort(list);
        }
}
```

نظرات خوانندگان

نویسنده: علي
تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۲:۴۶

با سلام؛ لطفا کلاس آخری را بیشتر توضیح دهید.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۲:۵۵

کلاس آخری با یک پیاده سازی عمومی کار می‌کنه. دیگه نمی‌دونه نحوه مرتب سازی چطور پیاده سازی شده. فقط می‌دونه یک متد Sort هست که دراختیارش قرار داده شده. حالا شما راحت می‌تونی الگوریتم مورد استفاده رو عوض کنی، بدون اینکه نیاز داشته باشی کلاس آخری رو تغییر بدی. باز هست برای توسعه. بسته است برای تغییر. به این نوع طراحی رعایت open closed principle هم می‌گن.

نویسنده: SB
تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۴:۲۳

بنظر شما متد Sort کلاس اولیه، نباید از نوع Virtual باشد؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۴:۴۸

نوع کلاسش abstract هست.

نویسنده: مجتبی شاطری
تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۶:۴۷

در صورتی از virtual استفاده می‌کنیم که یک پیاده سازی از متد Sort در SortStrategy داشته باشیم ، اما در اینجا طبق فرموده دوستمون کلاس ما فقط انتزاعی (Abstract) هست.

نویسنده: سید ایوب کوکبي
تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۱:۲۲

چرا استراتژی توسط Abstract پیاده سازی شده و از اینترفیس استفاده نشده؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۲:۵۱

تفاوت مهمی نداره ؛ فقط اینترفیس ورژن پذیر نیست. یعنی اگر در این بین متدی رو به تعاریف اینترفیس خودتون اضافه کردید، تمام استفاده کننده‌ها مجبور هستند اون رو پیاده سازی کنند. اما کلاس Abstract می‌تونه شامل یک پیاده سازی پیش فرض متد خاصی هم باشه و به همین جهت ورژن پذیری بهتری داره.
بنابراین کلاس Abstact یک اینترفیس است که می‌تواند پیاده سازی هم داشته باشد.
همین مساله خاص نگارش پذیری، در طراحی ASP.NET MVC به کار گرفته شده: (^)
برای من نوعی شاید این مساله اهمیتی نداشته باشه. اگر من قرارداد اینترفیس کتابخانه خودم را تغییر دادم، بالاخره شما با یک حداقل نق زدن مجبور به به روز رسانی کار خودتان خواهید شد. اما اگر مایکروسافت چنین کاری را انجام دهد، هزاران نفر شروع خواهند کرد به بد گفتن از نحوه مدیریت پروژه تیم‌های مایکروسافت و اینکه چرا پروژه جدید آن‌ها با یک نگارش جدید MVC کامپایل نمی‌شود. بنابراین انتخاب بین این دو بستگی دارد به تعداد کاربر پروژه شما و استراتژی ورژن پذیری قرار دادهای کتابخانه‌ای که ارائه می‌دهید.

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۳:۲۷

اطلاعات خوبی بود، ممنون، ولی با توجه به تجربه تون، در پروژه‌های متن باز فعلی تحت بستر دات نت بیشتر از کدام مورد استفاده میشه؟ اینترفیس روحیه نظامی خاصی به کلاس‌های مصرف کننده اش میده، یه همین دلیل من زیاد رقبت به استفاده از اون ندارم، آیا مواردی هست که چاره ای نباشه حتما از یکی از این دو نوع استفاده بشه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۳:۴۹

- اگر پروژه خودتون هست، از اینترفیس استفاده کنید. تغییرات آن و نگارش‌های بعدی آن تحت کنترل خودتان است و build دیگران را تحت تاثیر قرار نمی‌دهد.
- در پروژه‌های سورس باز دات نت، عموما از ترکیب این دو استفاده می‌شود. مواردی که قرار است در اختیار عموم باشند حتی دو لایه هم می‌شوند. مثلا در MVC یک اینترفیس IController هست و بعد یک کلاس Abstract به نام Controller، که این اینترفیس را پیاده سازی کرده برای ورژن پذیری بعدی و کنترلرهای پروژه‌های عمومی MVC از این کلاس Abstract مشتق می‌شوند یا در پروژه RavenDB از کلاس‌های Abstract زیاد استفاده شده، مانند AbstractIndexCreationTask و AbstractMultiMapIndexCreationTask و غیره.

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۷/۰۱ ۱۶:۱۱

توابع abstract بطور ضمنی virtual هستند.

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۷/۰۱ ۱۸:۳۸

در کلاس abstract نیز می‌توان از پیاده سازی پیشفرض استفاده کرد . یکی از تفاوتهای کلاس abstract با Interface همین ویژگی است که سبب ورژن پذیری آن شده است.

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۸/۲۱ ۹:۱۵

بهتر نیست در کلاس SortedList برای مشخص کردن استراتژی مرتب سازی، از روش تزریق وابستگی - Dependency Injection - استفاده بشه؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۸/۲۱ ۹:۲۵

خوب، الان هم وابستگی کلاس یاد شده از طریق سازنده آن در اختیار آن قرار گرفته و داخل خود کلاس وهله سازی نشده. (در این مطلب طراحی بیشتر مدنظر هست تا اینکه حالا این وابستگی به چه صورتی و کجا قرار هست وهله سازی بشه و در اختیار کلاس قرار بگیره؛ این مساله ثانویه است)

نویسنده: جمشیدی فر
تاریخ: ۱۳۹۲/۰۸/۲۱ ۱۱:۴۳

از طریق سازنده کلاس SortedList؟ بنظر نمیاداز طریق سازنده انجام شده باشه. ولی ظاهرا این امکان هست که کلاس بالادستی که می‌خواهد از SortedList استفاده کند، بتواند از طریق تابع SetSortStrategy کلاس مورد نظر رادر اختیار SortedList قراردهد. به نظر شبیه Setter Injection می‌شود.