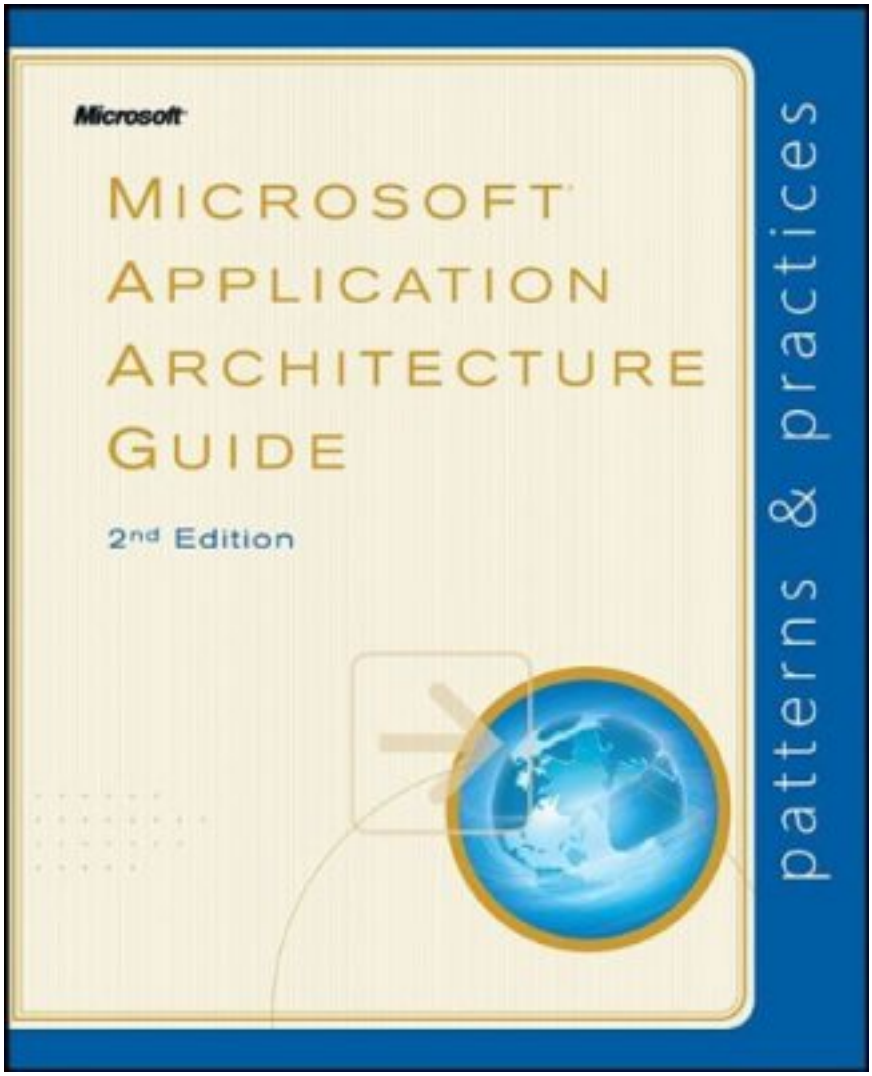
عنوان: **معرفی کتاب Microsoft Application Architecture Guid**
نویسنده: نیلوفر نوروزی
تاریخ: ۱۳۹۱/۰۳/۲۹ ۱۲:۱۵
آدرس: www.dotnettips.info
برچسب‌ها: Architecture, ebook

این کتاب شامل راهنمایی‌های عملی در استفاده از بهترین شیوه برای کاربرد معماری و طراحی الگوهاست.



اطلاعات بیشتر در اینجا

نظرات خوانندگان

نویسنده: Mohammad
تاریخ: ۱۳۹۱/۰۳/۲۹ ۱۳:۵۹

خیلی خوشحال شدم از اینکه این سایت دوباره شروع به کار کرد.

نویسنده: مهدی
تاریخ: ۱۳۹۱/۰۳/۲۹ ۱۴:۱۹

سلام.
اسم این کاری که شما انجام داده اید، "معرفی کتاب" نیست؛ لینک دادن به مطالب سایت‌های دیگه هستش. معرفی کتاب، شامل بخش‌های مختلفی میشه که از اون جمله میشه به مطالب زیر اشاره کرد:
دادن یک شمای کلی در مورد مطالب کتاب، نویسنده و ...
ذکر نام و پیش زمینه ای در مورد مولفین کتاب و تجارب هر یک
نوشتن چند پاراگراف در مورد مطالب کتاب و اینکه چرا خوندن این کتاب میتونه برای من نوعی مفید باشه
و ...

در غیر اینصورت، این یه Tweet محسوب میشه و عنوان "معرفی کتاب"، عنوان سنگینی برای چنین لینک هایی محسوب میشه.

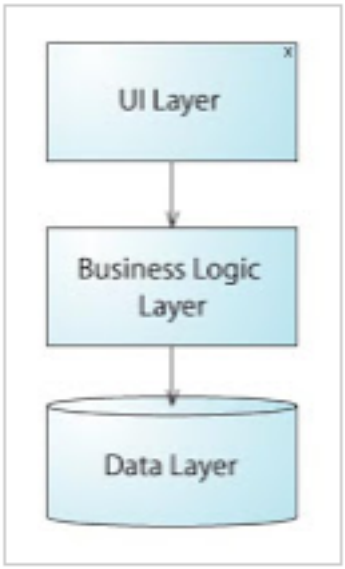
ممنون و موفق باشید.

نویسنده: محمد صادق شاد
تاریخ: ۱۳۹۱/۰۳/۳۰ ۱۴:۳۴

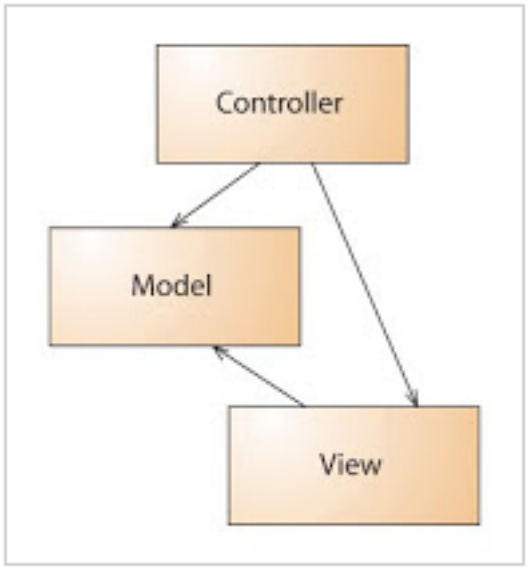
آره مهدی درست میگه. برای لینک دادن هم این قسمت در نظر گرفته شده
http://www.dotnettips.info/DailyLinks

عنوان: MVC vs 3-Tier Pattern
نویسنده: مهرداد اشکانی
تاریخ: ۱۳۹۲/۰۳/۲۶ ۱۴:۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, MVC, Architecture, معماری سه لایه, MVC معماری

من تا به حال برنامه نویس‌های زیادی را دیده‌ام که می‌پرسند «چه تفاوتی بین الگوهای معماری MVC و Three-Tier وجود دارد؟» قصد من روشن کردن این سردرگمی، بوسیله مقایسه هردو، با کنار هم قرار دادن آنها می‌باشد. حداقل در این بخش ، من اعتقاد دارم، منبع بیشتر این سردرگمی‌ها در این است که هر دوی آنها، دارای سه لایه متمایز و گره، در دیاگرام مربوطه‌اشان هستند.



Three-Tier



MVC

اگر شما به دقت به دیاگرام آنها نگاه کنید، پیوستگی را خواهید دید. بین گره‌ها و راه اندازی آنها، کمی تفاوت است.

معماری سه لایه

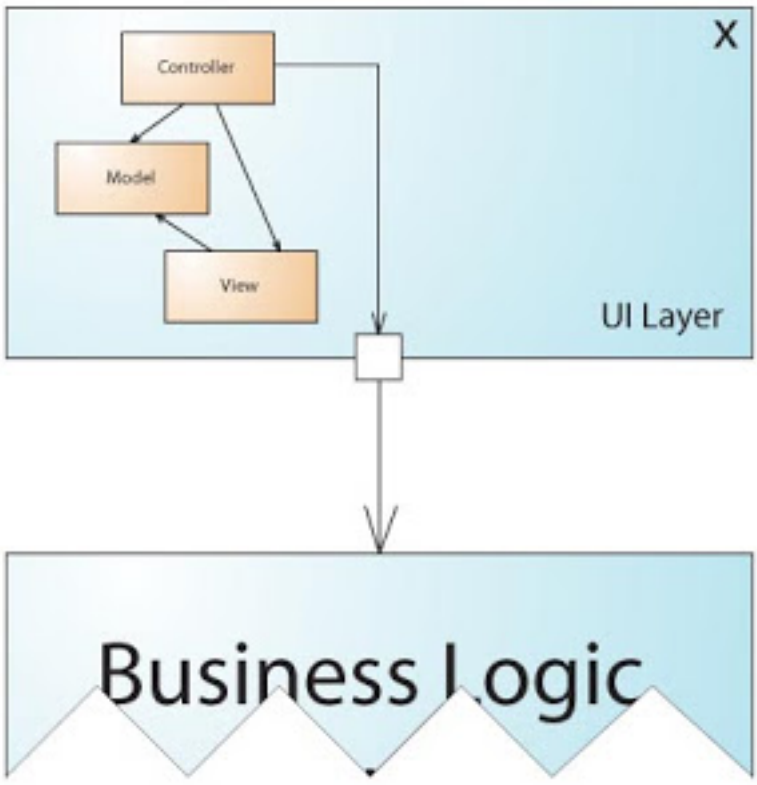
سیستم‌های سه لایه، واقعاً لایه‌ها را می‌سازند: لایه UI به لایه Business logic دسترسی دارد و لایه Business logic به لایه Data دسترسی دارد. اما لایه UI دسترسی مستقیمی به لایه Data ندارد و باید از طریق لایه Business logic و روابط آنها عمل کند. بنابراین می‌توانید فکر کنید که هر لایه، بعنوان یک جزء، آزاد است؛ همراه با قوانین محکم طراحی دسترسی بین لایه ها.

MVC

در مقابل، این Pattern ، لایه‌های سیستم را نگهداری نمی‌کند. کنترلر به مدل و View (برای انتخاب یا ارسال مقادیر) دسترسی دارد. View نیز دسترسی دارد به مدل . دقیقاً چطور کار می‌کند؟ کنترلر در نهایت نقطه تصمیم گیری منطقی است. چه نوع منطقی؟ نوعاً، کنترلر، ساخت و تغییر مدل را در اکشن‌های مربوطه، کنترل خواهد کرد . کنترلر سپس تصمیم گیری می‌کند که برای منطق داخلیش، کدام View مناسب است. در آن نقطه، کنترلر مدل را به View ارسال می‌کند. من در اینجا چون هدف بحث مورد دیگه‌ای می‌باشد، مختصر توضیح دادم.

چه موقع و چه طراحی را انتخاب کنم؟

اول از همه، هر دو طراحی قطعاً و متقابلاً منحصر بفرد نیستند. در واقع طبق تجربه‌ی من، هر دو آنها کاملاً هماهنگ هستند. اغلب ما از معماری چند لایه استفاده می‌کنیم مانند معماری سه لایه، برای یک ساختار معماری کلی. سپس من در داخل لایه UI ، از MVC استفاده می‌کنم، که در زیر دیاگرام آن را آورده ام.



نظرات خوانندگان

نویسنده: محسن اسماعیل پور
تاریخ: ۱۳۹۲/۰۳/۲۶ ۲۲:۳۳

3-Layer در واقع Architecture Style هست اما MVC یک Design Pattern هست پس مقایسه مستقیم نمیدونم کاری دست باشد یا نه اما میتونیم به این شکل نتیجه گیری کنیم:
Data Access: شامل کلاسهای ADO.NET یا EF برای کار با دیتابیس.
Business Logic: یا همان Domain logic که میتوان Model رو به عنوان Business entity در این لایه بکار برد.
UI Layer: بکارگیری Controller و View در این لایه

نویسنده: یزدان
تاریخ: ۱۳۹۲/۰۳/۲۷ ۱:۳۳

در برنامه نویسی 3لایه کار Business Logic به طور واضح و شفاف چی هست و چه کارهایی در این لایه لحاظ میشه ؟

نویسنده: fss
تاریخ: ۱۳۹۲/۰۳/۲۷ ۸:۴۱

منم یه مدت دچار این ابهام بودم. ولی الان اینطور نتیجه می‌گیرم:

mvc کلا در لایه UI قرار داره. یعنی اگر شما لایه BL و DAL رو داشته باشید، حالا میتونید لایه UI رو با یکی از روش ها، مثلا سیلورلایت، asp.net mvc یا asp.net web form پیاده کنید.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۳/۲۷ ۸:۴۹

همون لایه UI هم نیاز به جداسازی کدهای نمایشی از کدهای مدیریت کننده آن برای بالابردن امکان آزمایش کردن و یا حتی استفاده مجدد قسمت‌های مختلف اون داره. در این حالت شما راحت نمی‌تونید MVC و Web forms رو در یک سطح قرار بدی (که اگر اینطور بود اصلا نیازی به MVC نبود؛ نیازی به MVVM برای سیلورلایت یا WPF نبود و یا نیازی به MVP برای WinForms یا Web forms نبود).

نویسنده: fss
تاریخ: ۱۳۹۲/۰۳/۲۷ ۹:۱۳

دوست عزیز من متوجه منظور شما نشدم. حرف من اینه که MVC، MVVM، MVP و .. در سطح UI پیاده میشن.

نویسنده: مهرداد اشکانی
تاریخ: ۱۳۹۲/۰۳/۲۷ ۹:۱۸

لایه business Logic در واقع لایه پیاده سازی Business پروژه شما می‌باشد با یک مثال عرض می‌کنم فرض کنید در لایه UI شما لازم دارید یک گزارش از لیست مشتریانی که بالاترین خرید را در 6 ماه گذشته داشته اند و لیست تراکنش مالی آنها را بدست آورید.برای این مورد شما توسط کلاسهای و متدهای لازم ، در لایه Business Logic این عملیات را پیاده سازی می‌کنید.

نویسنده: مهرداد اشکانی
تاریخ: ۱۳۹۲/۰۳/۲۷ ۹:۳۸

این طور نیست دوست عزیز شما می‌تونید حتی برای Model هم لایه در نظر بگیرید که براحتی توسط لایه Business و کلا لایه‌های دیگر در دسترس باشد.که این مورد الان در MVC خیلی کاربرد دارد.مواردی که من عرض کردم برای رفع ابهام بین معماری چند لایه و Pattern MVC بود.

نویسنده: داود
تاریخ: ۱۳۹۲/۰۳/۲۷ ۱۷:۱۴

به بنظر بنده هم معماری رو نمی‌شود با الگو مقایسه کرد به هر حال خود الگوی mvcها یک سری لایه داره و تا اونجایی هم که می‌دونم فرق tier با layer اینه که tierها رو از لحاظ فیزیکی هم جدا می‌کنند

نویسنده: وحید فرهمندیان
تاریخ: ۱۳۹۳/۱۲/۰۴ ۱۷:۱۶

لایه کسب و کار مغز برنامه شما میباشد. یک زمانی میخواهید معادله ریاضی حل کنید در این لایه و زمانی نیز نیاز است مقداری داده از انباره داده خود بخوانید. لذا UI درخواست محاسبه معادله یا استخراج گزارش را به کسب و کار میدهد، کسب و کار بررسی میکند تا درخواست را پاسخ دهد. اگر برای پاسخ نیاز به انباره داده بود به لایه داده میفرستد تا مطابق با آن درخواست داده‌های مناسب استخراج شده و برگشت داده شوند.
نکته ای که وجود دارد این است که لایه داده حتما نباید با یک پایگاه داده ارتباط برقرار کند، و لایه UI نیز نباید شخصا کار پردازشی یا منطقی انجام دهد و این کارها باید به لایه کسب و کار ارجاع داده شوند.

نویسنده: ریوف مدرسی
تاریخ: ۱۳۹۴/۰۴/۰۵ ۱۸:۳۲

به نظر من این درست نیست که بگیم mvc کلا در لایه UI می‌باشد، ببینید شما وقتی می‌خواهید یک application را توسعه بدید علاوه بر اهداف بیزینسی application که قصد توسعه ان را دارید یکسری اهداف تکنولوژیکی هم مد نظر دارید، حالا بعضی از این اهداف می‌تواند بسیار پایه ای باشد مانند اینکه شما این application را بصورت web app , win app یا mobile app یا ترکیبی پیاده سازی کنید، وبعضی تصمیمات هم بعد از این تصمیم گیری انجام میشوند، برای مثال شما در نظر می‌گیرید که می‌خواهید این application را به صورت یک web app پیاده سازی کنید، حال شما ممکن است به عنوان طراح نرم افزار یا یک معمار یکسری concern ریز و درشت دیگر برایتان ایجاد شود، باز هم برای مثال: این application باید SOC را در تمامی سطوح درنظر بگیرد و decoupling به یک سطح مناسب برساند، ویا اینکه هدف این که لایه business را طوری طراحی کنیم که به یک repository خاص یا یک ui framework وابسته نباشد، اما هدف من از این همه اسمان ریسمان‌ها این بود که بگم که مثلا نمی‌توان گفت که فلان تکنولوژی باید در فلان موقعیت و به فلان روش استفاده شود بلکه این شما هستید که بر حسب concernهای خود چگونه تصمیم بگیرید.

عنوان: **تفاوت Desktop Application با Web Application**
نویسنده: یاسر مرادی
تاریخ: ۱۳۹۳/۰۴/۲۰ ۱۲:۴۵
آدرس: www.dotnettips.info
گروه‌ها: Architecture, Design

در هنگام گفتگو با افراد مختلفی که در پروژه‌های توسعه نرم افزار، نقش‌های مختلفی را دارا می‌باشند، یکی از جالب‌ترین و اساسی‌ترین بحث‌ها تفاوت بین Desktop App و Web App می‌باشد، و این که پروژه بر اساس کدام مدل باید نوشته شود.

در اینترنت و در منابع معتبر، تفسیرهای متفاوتی از این دو وجود دارد، که گاه دقیقا با نظر من یکی بوده و گاه تا 180 درجه بر عکس هستند، آنچه که در ادامه می‌خوانید می‌تواند لزوما نظر شما نباشد.
گروهی از افراد بر این باور هستند که اجرای برنامه در محیط مرورگر (ظاهر مرورگر و نه Sandbox آن)، یکی از ملاک‌های ما بین Desktop App و Web App است، گروهی دیگر نیز اجرا شدن برنامه بر روی بستر اینترنت و یا شبکه‌ی محلی را جزو ملاک‌ها می‌دانند، و گروهی دیگر نیز زبان برنامه نویسی برنامه را ملاک می‌دانند، برای مثال اگر با HTML/JS باشد Web App است، اگر نه Desktop App است.
اما آنچه که در عمل می‌تواند تفاوت بین یک Desktop App را با یک Web App مشخص کند، **رفتار و عملکرد خود آن برنامه** است، نه بستر اجرای آن و این که آن رفتار منتج شده از چه کدی و چه زبان برنامه نویسی ای است.
اگر کمی دقیق به مطلب نگاه کنیم، می‌بینیم این که یک برنامه در چارچوب ظاهری یک مرورگر (نه Sandbox آن) اجرا شود، اصلا مقوله ای اهمیت دار نیست، کما این که برای مثال Silverlight اجازه می‌دهد، برنامه هم در داخل مرورگر و هم در بیرون از آن اجرا شود، و این کار با یک کلیک امکانپذیر است، آیا با همین یک کلیک برنامه از Web App به Desktop App تبدیل می‌شود یا بالعکس ؟
آیا یک برنامه مبتنی بر دلفی که تا همین یک ساعت پیش بر روی شبکه محلی در حال اجرا بوده، با انتقال پیدا کردن آن بر روی شبکه‌ی اینترنت، تبدیل به یک Web App می‌شود؟
آیا اگر ما با HTML/JS یک برنامه Native برای ویندوز فون بنویسیم که تک کاربره آفلاین باشد و اصلا سروری هم نداشته باشد، آیا Web App نوشته ایم ؟ **اصلی‌ترین** تفاوت مابین Web App و Desktop App که به **تفاوت در عملکرد آنها و مزایا و معایب آنها** منجر می‌شود، این است که انجام کارهایی که اپراتور با آنها در سمت کلاینت و سیستم مشتری سر و کار دارد، در کجا صورت می‌پذیرد؟
برای مثال در نظر بگیرید که یک دیتاگرید داریم که دارای Paging است، و ما از Page اول به Page بعدی می‌رویم، در یک Desktop App تنها **اطلاعات** از سرور گرفته می‌شود، و ترسیم خطوط و ستون‌ها و ردیف‌ها و ظاهر نمایشی دیتاگرید بر عهده کلاینت است، برای مثال اگر ستون قیمت داشته باشیم، و بخواهیم برای ردیف هایی که قیمت آنها زیر 10000 ریال است، قیمت به شکل سبز رنگ نمایش داده شود و برای بقیه ردیف‌ها به رنگ قرمز باشد، پردازش این مسئله و این if به عهده کلاینت است، اما در یک Web App، علاوه بر اطلاعات، تعداد زیادی tagهای مختلف، مانند table - tr - td و ... نیز به همراه اطلاعات آورده می‌شوند، که وظیفه نمایش ظاهری اطلاعات را بر عهده دارند، و آن if مثال ما یعنی رنگ سبز و قرمز در سمت سرور مدیریت شده است، و کلاینت در اینجا نمایش دهنده‌ی آن چیزی است که به صورت آماده از سرور آورده شده است.
در برنامه‌های Desktop آنچه که در سمت سرور وجود دارد، برای مثال یک WCF Service یا ASP.NET Web API است که فقط به رد و بدل کردن اطلاعات می‌پردازد، اما در Web Appها در سمت سرور ASP.NET MVC، ASP.NET Web Forms و PHP وجود دارند که علاوه بر اطلاعات برای کلاینت شما ظاهر صفحات را نیز آماده می‌کنند، و ظاهر اصلی صفحات از سمت سرور به سیستم مشتری ارسال می‌شوند، اگر چه که ممکن است در سمت کلاینت تغییراتی را داشته باشند.
به هر میزان رفتار برنامه ما شبیه به حالت اول باشد، برنامه ما Desktop App بوده و به هر میزان برنامه ما به حالت دوم نزدیک‌تر باشد، برنامه ما Web App است.
مزیت اصلی Web Appها در عدم انداختن بار زیاد بر روی دوش کلاینت‌های بعضا نحیف بوده، و عملا کلاینت به علت این که کار خاصی را انجام نمی‌دهد، پیش نیاز نرم افزاری و یا سخت افزاری خاصی احتیاج ندارد، و این مورد Web Appها را به یک گزینه ایده آل برای وب سایت هایی تبدیل کرده است که با عموم مردم در ارتباطند، زیرا که امکان ارائه آسان برنامه وجود دارد و تقریبا همه می‌توانند از آن استفاده کنند.
با توجه به شناخت عموم از برنامه‌های Web App به توضیح بیشتر برنامه‌های Desktop App می‌پردازم.
مزیت اصلی Desktop Appها در سرعت عمل بالاتر(به علت این که فقط دیتا را رد و بدل می‌کند)، توانایی بیشتر در استفاده از منابع سیستمی مانند سرویس نوشتن، و امکانات محلی مانند ارائه Notification و ... است، و در کنار آن برای مثال یک Desktop App

می‌تواند به نحوی طراحی شود که به صورت Offline نیز کار کند.
این مزیت‌ها باعث می‌شود که Desktop Appها گزینه ای مناسب برای برنامه‌های سازمانی باشند.
ضعفی که از گذشته در Desktop Appها وجود داشته است، که البته به معماری Desktop App بر نمی‌گردد، بلکه متاثر از امکانات است، عدم Cross Platform بودن آنها بوده است، تا آنجا که Desktop App در نظر خیلی از افراد همان نوشتن برنامه برای سیستم عامل ویندوز است.
با توجه به رویکرد جدی ای که در طول دو سال اخیر برای نوشتن برنامه Desktop App به شکل Cross Platform رخ داده است، خوشبختانه این مشکل حل شده است و اکنون لااقل دو راهکار جدی برای نوشتن یک برنامه Cross Platform با ویژگی‌های Desktop وجود دارد، که یکی از آنها راه حل‌های مبتنی بر HTML/JS است و دیگری راه حل‌های مبتنی بر C#/XAML
در راه حل‌های مبتنی بر HTML/JS در صورتی که شما برنامه را به شکل Web App طراحی نکرده باشید، و برای مثال در آن از ASP.NET Web Forms و ASP.NET MVC، PHP و ... استفاده نکرده باشید، می‌توانید یک خروجی کاملا Native با تمامی ویژگی‌های Desktop App برای انواع پلتفرم‌ها بگیرید.
استفاده از فریم ورک هایی که با طراحی Desktop App سازگار هستند، مانند Angular JS، Kendo UI و Jay-data، Ext JS و ... و استفاده از مدل طراحی Single Page Application می‌تواند سیستم کدنویسی ای ساده را فراهم آورد، که در آن شما با یک بار نوشتن برنامه می‌توانید خروجی اکثر پلتفرم‌های مطرح را داشته باشید، اعم از ویندوز فون، اندروید، iOS و ویندوز
امروزه حتی مرورگرها با فراهم آوردن امکاناتی مانند Client side databases و Manifest based deployment اجازه نوشتن برنامه Desktop با HTML/JS را که حتی می‌تواند Offline کار کند را به شما ارائه می‌کنند.
در کنار این راهکار، استفاده از C#/XAML برای نوشتن برنامه برای اکثر پلتفرم‌های مطرح بازار اعم از اندروید، iOS و Windows Phone و ویندوز، نیز به عنوان راهکاری دیگر قابلیت استفاده را دارا است.
حرکت پر شتاب و پر انرژی جهانی برای توسعه Cross Platform Desktop Development، خوشبختانه توانسته است تا حد زیادی امتیاز نوشتن برنامه‌های Desktop را در سیستم‌های Enterprise بالا ببرد.

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۴/۲۰ ۱۳:۵

مطلبی هم من چند سال قبل در این مورد نوشته بودم
« چرا در سازمان‌ها برنامه‌های وب جایگزین برنامه‌های دسکتاپ شده‌اند (یا می‌شوند)؟ »

نویسنده: علیرضا
تاریخ: ۱۳۹۳/۰۴/۲۱ ۱۰:۱۱

به نظر من این بحث به همین سادگی نیست و انتخاب پلتفرم اجرای پروژه به پارامترها و ویژگی‌های زیادی مرتبط هست. بطور مثال سرعت توسعه برنامه‌های ویندوز حداقل در قسمت طراحی رابط کاربری سریعتر و ساده‌تر از وب هست. و یا در مثال دیگر رفتار غیر یکسان مرورگرها مشکلاتی را در طراحی نرم افزارهای بزرگ ایجاد می‌کنه و مشکل ساز میشه من بعد از سال‌ها طراحی سیستم‌های سازمانی روش استفاده ترکیبی از پلتفرم‌های مختلف را انتخاب کردم بطور مثال قسمت مدیریت یک سیستم را بصورت ویندوزی و قسمت رابط ماربری را با وب و... طراحی کردم. متاسفانه طراحی اولیه زبان HTML با هدف نمایش اطلاعات بوده و بهبودهای اخیر از جمله وب 2 پاسخی منطقی به نیاز به توسعه نرم افزارهای Cross Platform بوده ولی هنوز هم با پیچیدگی‌های زیادی روبروست. به نظر قابلیت‌های نرم افزار تحت وب بیش از واقعیت بزرگ نمایی شده و هنوز هم در برخی راه کارها استفاده از نرم افزارهای تحت ویندوز گزینه مناسب‌تری خواهد بود اما این به معنی چشم پوشی بر مزایای منحصر به فرد وب نخواهد بود و هنوز انتخاب پلتفرم بستگی زیادی به نیازمندی‌ها و امکانات پروژه خواهد داشت.

نویسنده: رحیم
تاریخ: ۱۳۹۳/۰۴/۲۱ ۲۲:۴۴

سلام لطفا در مورد این جمله بیشتر توضیح دهید
در کنار این راهکار، استفاده از C#/XAML برای نوشتن برنامه برای اکثر پلتفرم‌های مطرح بازار اعم از اندروید، iOS و Windows Phone و ویندوز، نیز به عنوان راهکاری دیگر قابلیت استفاده را دارا است.
آیا منظور شما استفاده از نرم افزارهای شرکت ثالث نظیر xamarin و .... می‌باشد یا تکنولوژی جدیدی از سمت مایکروسافت ارائه شده که این امکان رو میسر می‌کنه

نویسنده: یاسر مرادی
تاریخ: ۱۳۹۳/۰۴/۲۲ ۸:۴۳

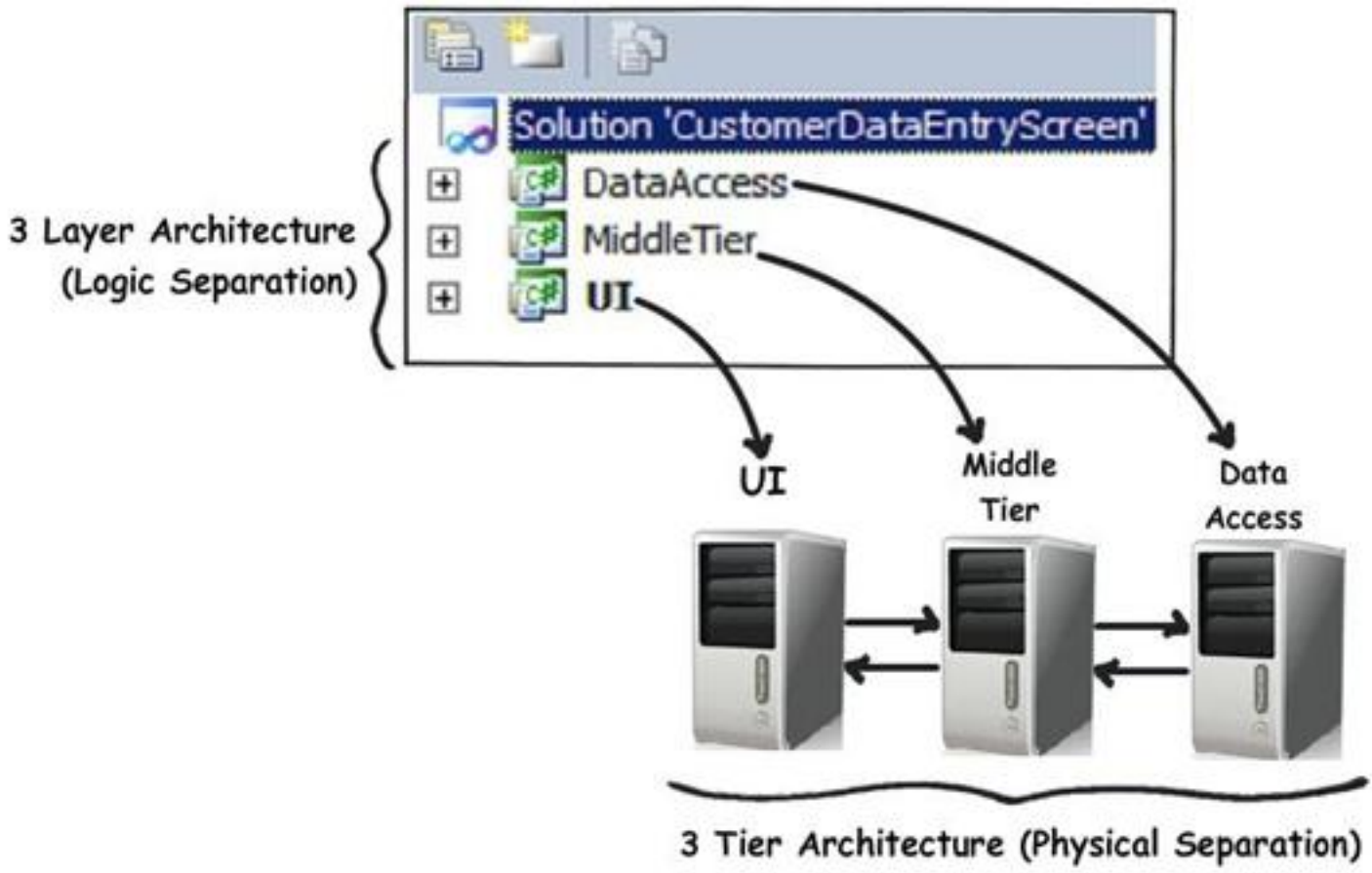
بله، منظور روش‌های ارائه شده مبتنی بر پلتفرم Xamarin است، البته در نظر بگیرید این کار بدون کمک‌های فنی ارائه شده توسط مایکروسافت و همچنین رفع مشکل لایسنس Portable Class Libraryها و ... از سوی مایکروسافت امکانپذیر نبود.
مایکروسافت بنظر قصد پشتیبانی فنی و مالی و در نهایت خرید Xamarin رو داره، و بنظر نمی‌آد که بخواد این مسیر رو از نو پیش بره، چون واقعا کار زیادی می‌بره

عنوان: **تبادل داده ها بین لایه ها- قسمت اول**
نویسنده: ریوف مدرسی
تاریخ: ۱۳۹۴/۰۳/۳۱ ۲۱:۳۰
آدرس: www.dotnettips.info
گروه‌ها: ADO.NET, Design patterns, Architecture, OOP, N-Layer Architecture

معماری لایه بندی شده، یک معماری بسیار همه گیر می‌باشد. به این خاطر که به راحتی decoupling ، SOC و قدرت درک کد را بسیار بالا می‌برد. امروزه کمتر برنامه نویس و فعال حوضه‌ی نرم افزاری است که با لایه‌های کلی و وظایف آنها آشنا نباشد (UI layer آنچه که ما می‌بینیم، middle layer برای مقاصد منطق کاری، data access layer برای هندل کردن دسترسی به داده‌ها). اما مسئله ای که بیشتر برنامه نویسان و توسعه دهندگان نرم افزار با استانداردهای آن آشنا نیستند، راه‌های تبادل داده‌ها مابین layer ها می‌باشد. در این مقاله سعی داریم راه‌های تبادل داده‌ها را مابین لایه‌ها، تشریح کنیم.

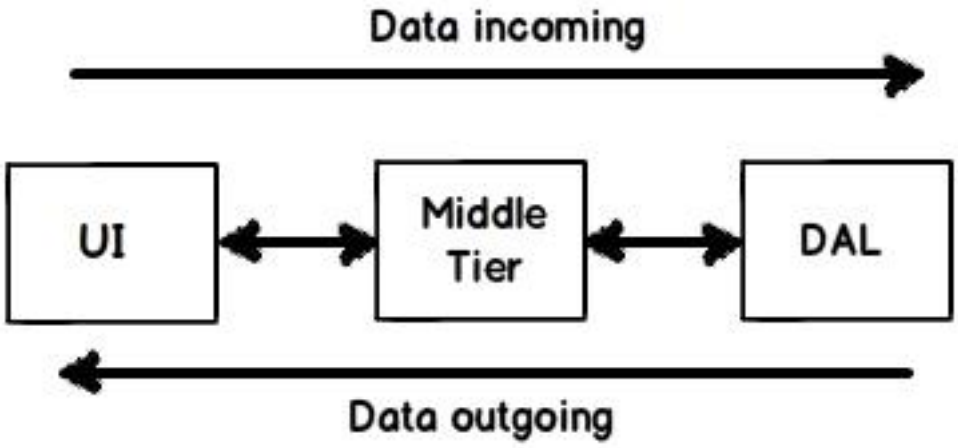
Layers و Tiers با هم متفاوت هستند

Layer با Tier متفاوت است. هنگامیکه در مورد مفهوم layer و Tier دچار شک شدید، دیاگرام ذیل می‌تواند به شما بسیار کمک کند. layer به مجزاسازی منطقی کد و Tier هم به مجزا سازی فیزیکی در ماشین‌های مختلف اطلاق می‌شود. توجه داشته باشید که این نکته یک شفاف سازی کلی در مورد یک مسئله مهم بود.



داده‌های وارد شونده(incoming) و خارج شونده(outgoing)

ما باید تبادل داده‌ها را از دو جنبه مورد بررسی قرار دهیم؛ اول اینکه داده‌ها چگونه به سمت لایه Data Access می‌روند، دوم اینکه داده‌ها چگونه به لایه UI پاس می‌شوند، در ادامه شما دلیل این مجزا سازی را درک خواهید کرد.



روش اول: Non-uniform

این روش اولین روش و احتمالا عمومی‌ترین روش می‌باشد. خوب، اجازه دهید از لایه‌ی UI به لایه DAL شروع کنیم. داده‌ها از لایه UI به Middle با استفاده از getter ها و setter ها ارسال خواهد شد. کد ذیل این مسئله را به روشنی نمایش می‌دهد.

```
Customer objCust = new Customer();
objCust.CustomerCode = "c001";
objCust.CustomerName = "Shivprasad";
```

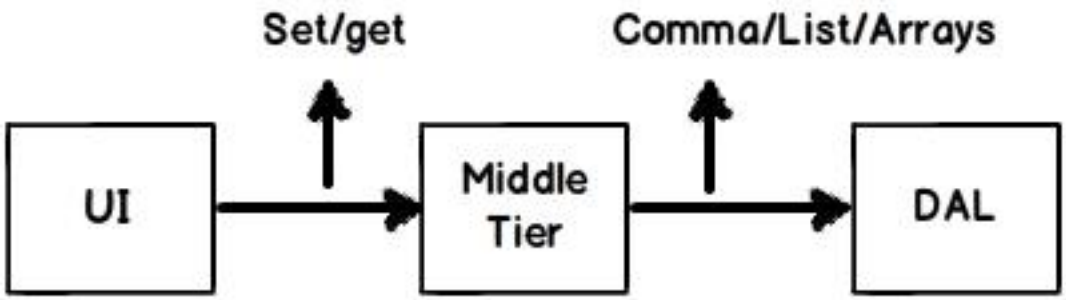
بعد از آن، از لایه Middle به لایه Data Access داده‌ها با استفاده از مجزاسازی به وسیله comma و آرایه‌ها و سایر روش‌های non-uniform پاس داده می‌شوند. در کد ذیل به متد Add دقت کنید که چگونه فراخوانی به لایه Data Access را با استفاده از پارامترهای ورودی انجام می‌دهد.

```
public class Customer
{
    private string _CustomerName = "";
    private string _CustomerCode = "";
    public string CustomerCode
    {
        get { return _CustomerCode; }
        set { _CustomerCode = value; }
    }
    public string CustomerName
    {
        get { return _CustomerName; }
        set { _CustomerName = value; }
    }
    public void Add()
    {
        CustomerDal obj = new CustomerDal();
        obj.Add(_CustomerName,_CustomerCode);
    }
}
```

کد ذیل، متد add در لایه Data Access را با استفاده از دو متد نمایش می‌دهد.

```
public class CustomerDal
{
    public bool Add(string CustomerName,string CustomerCode)
    {
        // Insert data in to DB
    }
}
```

بنابراین اگر بخواهیم به صورت خلاصه نحوه پاس دادن داده‌ها را در روش non-uniform بیان کنیم، شکل ذیل به زیبایی این مسئله را نشان می‌دهد.



- از لایه UI به لایه Middle با استفاده از setter و getter

- از لایه Middle به لایه data access با استفاده از array ، input ، comma

حال نوبت این است بررسی کنیم که چگونه داده‌ها از DAL به UI در روش non-uniform پاس خواهند شد. بنابراین اجازه دهید که اول از UI شروع کنیم. از لایه UI داده‌ها با استفاده از object های لایه Middle واکشی می‌شوند.

```
Customer obj = new Customer();
List<Customer> oCustomers = obj.getCustomers();
```

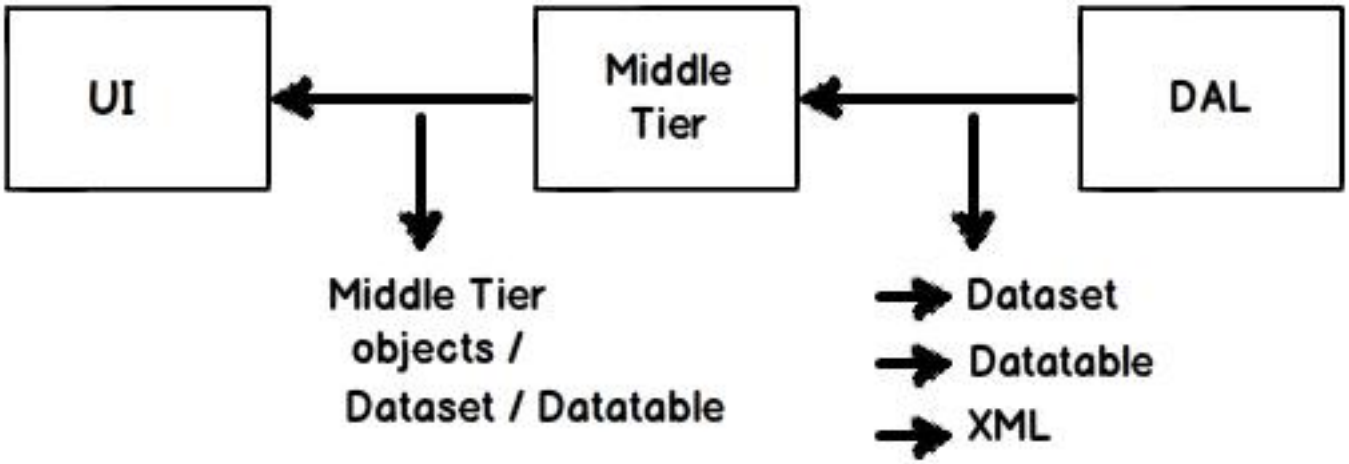
از لایه Middle هم داده‌ها با استفاده از datatable ، dataset و xml پاس خواهند شد. مهمترین مسئله برای لایه middle ، loop بر روی dataset و تبدیل آن به strong type object ها می‌باشد. برای مثال می‌توانید کد تابع getCustomers که بر روی ، dataset loop می‌زند و یک لیست از Customer ها را آماده می‌کند در ذیل مشاهده کنید. این تبدیل باید انجام شود، به این دلیل که UI به کلاس‌های strongly typed دسترسی دارد.

```
public class Customer
{
    private string _CustomerName = "";
    private string _CustomerCode = "";
    public string CustomerCode
    {
        get { return _CustomerCode; }
        set { _CustomerCode = value; }
    }
    public string CustomerName
    {
        get { return _CustomerName; }
        set { _CustomerName = value; }
    }
    public List<Customer> getCustomers()
    {
        CustomerDal obj = new CustomerDal();
        DataSet ds = obj.getCustomers();
        List<Customer> oCustomers = new List<Customer>();
        foreach (DataRow orow in ds.Tables[0].Rows)
        {
            // Fill the list
        }
        return oCustomers;
    }
}
```

با انجام این تبدیل به یکی از بزرگترین اهداف معماری لایه بندی شده می‌رسیم؛ یعنی اینکه « UI نمی‌تواند به طور مستقیم به کامپوننت‌های لایه Data Access مانند OLEDB ، ADO.NET و غیره دستیابی داشته باشد. با این روش اگر ما در ادامه متدولوژی Data Access را تغییر دهیم تاثیری بر روی لایه UI نمی‌گذارد.» آخرین مسئله اینکه کلاس CustomerDal یک Dataset را با استفاده از ADO.NET بر می‌گرداند و Middle از آن استفاده می‌کند.

```
public class CustomerDal
{
    public DataSet getCustomers()
    {
        // fetch customer records
        return new DataSet();
    }
}
```

حال اگر بخواهیم حرکت داده‌ها را به لایه UI ، به صورت خلاصه بیان کنیم، شکل ذیل کامل این مسئله را نشان می‌دهد.



· داده‌ها از لایه DAL به لایه Middle با استفاده از XML ، Datareader ، Dataset ارسال خواهند شد.

· از لایه Middle به UI از strongly typed classes استفاده می‌شود.

مزایا و معایب روش non-uniform

یکی از مزایای non-uniform

· به راحتی قابل پیاده سازی می‌باشد، در مواردی که روش data access تغییر نمی‌کند این روش کارآیی لازم را دارد.

تعدادی از معایب این روش

· به خاطر اینکه یک ساختار uniform نداریم، بنابراین نیاز داریم که همیشه در هنگام عبور از یک لایه به یک لایه‌ی دیگر از یک ساختار به یک ساختار دیگر تبدیل را انجام دهیم.

· برنامه نویسان از روش‌های خودشان برای پاس دیتا استفاده می‌کنند؛ بنابراین این مسئله خود باعث پیچیدگی می‌شود.

· اگر برای مثال شما بخواهید متدولوژی Data Access خود را تغییر دهید، تغییرات بر تمام لایه‌ها تاثیر می‌گذارد.

نظرات خوانندگان

نویسنده: بابک جهانگیری
تاریخ: ۱۳۹۴/۰۴/۰۴ ۱۳:۲۰

آیا در این روش می‌توان به صورت DataView لیست مشتریها رو برگردوند به جای اینکه از List<Customer> استفاده کنیم ؟ باز هم به آن non-uniform می‌گویند ؟

نویسنده: ریوف مدرسی
تاریخ: ۱۳۹۴/۰۴/۰۵ ۱۷:۵۳

در این روش مسئله اصلی این نیست که داده‌ها رو به صورت list یا Dataview برگردونید، بلکه مسئله اصلی این است که شما مجبورید در گذر از هر لایه تبدیل ساختار داده‌ها را انجام دهید، پس نکته این روش این است که تعداد تبدیل ساختار داده‌ها زیاد است.

نویسنده: محسن اسماعیل پور
تاریخ: ۱۳۹۴/۰۴/۰۸ ۸:۲۵

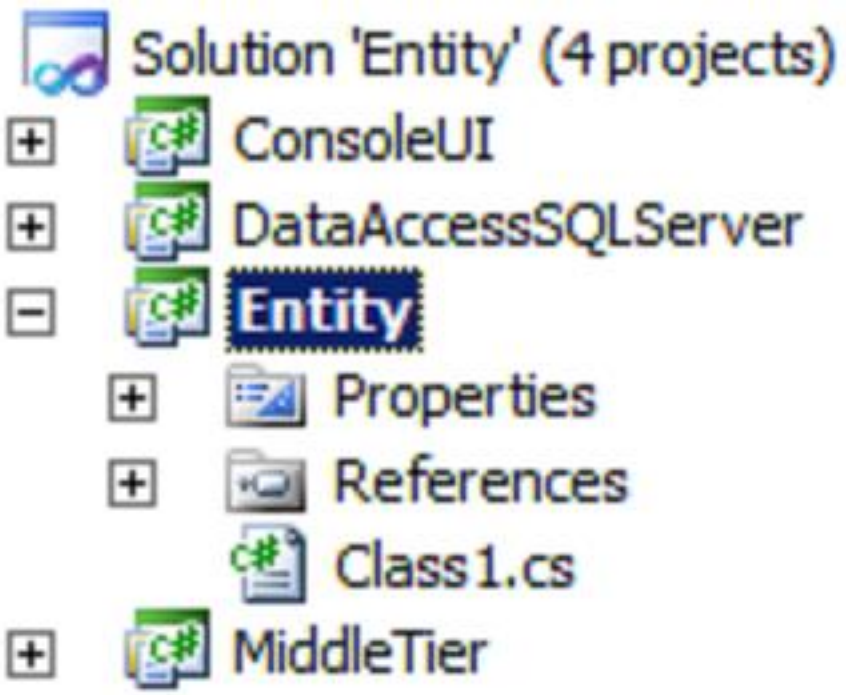
مدل Customer که شما برای مثالهایتان از آن استفاده کرده اید از Active record pattern تبعیت میکند. از آنجا که Entity یا Model با عملیات CRUD لایه دیتا Couple شده و بعضا ممکن است Business Logic داخل این متدها قرار گیرد، این مسئله با Sepration Of Concern منافات دارد.

عنوان: **تبادل داده ها بین لایه ها- قسمت دوم**
نویسنده: ریوف مدرسی
تاریخ: ۱۳۹۴/۰۴/۳۰ ۱:۲۵
آدرس: www.dotnettips.info
گروه‌ها: ADO.NET, Design patterns, Architecture, OOP, N-Layer Architecture

قسمت اول : تبادل داده‌ها بین لایه ها- قسمت اول

روش دوم: Uniform(Entity classes)

روش دیگر پاس دادن داده‌ها، روش uniform است. در این روش کلاس‌های Entity ، یک سری کلاس ساده به همراه یکسری Property های Get و Set می‌باشند. این کلاس‌ها شامل هیچ منطق کاری نمی‌باشند. برای مثال کلاس CustomerEntity که دارای دو Property ، Customer Name و Customer Code می‌باشد. شما می‌توانید تمام Entity ها را به صورت یک پروژه‌ی مجزا ایجاد کرده و به تمام لایه‌ها رفرنس دهید.




```
public class CustomerEntity
{
    protected string _CustomerName = "";
    protected string _CustomerCode = "";
    public string CustomerCode
    {
        get { return _CustomerCode; }
        set { _CustomerCode = value; }
    }
    public string CustomerName
    {
        get { return _CustomerName; }
        set { _CustomerName = value; }
    }
}
```

خوب، اجازه دهید تا از CustomerDal شروع کنیم. این کلاس یک Collection از CustomerEntity را بر می‌گرداند و همچنین یک CustomerEntity را برای اضافه کردن به دیتابیس . توجه داشته باشید که لایه Data Access وظیفه دارد تا دیتای دریافتی از دیتابیس را به CustomerEntity تبدیل کند.

```
public class CustomerDal
{
    public List<CustomerEntity> getCustomers()
    {
        // fetch customer records
        return new List<CustomerEntity>();
    }
    public bool Add(CustomerEntity obj)
    {
        // Insert in to DB
        return true;
    }
}
```

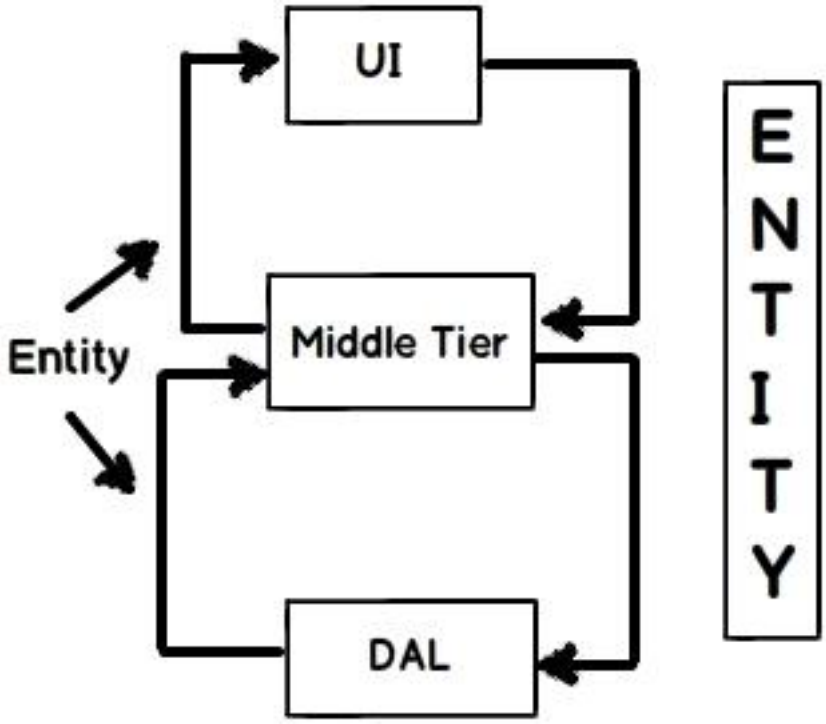
لایه Middle از CustomerEntity ارث بری می‌کند و یکسری operation را به entity class اضافه خواهد کرد. داده‌ها در قالب Entity Class به لایه Data Access ارسال می‌شوند و در همین قالب نیز بازگشت داده می‌شوند. این مسئله در کد ذیل به روشنی مشاهده می‌شود.

```
public class Customer : CustomerEntity
{

    public List<CustomerEntity> getCustomers()
    {
        CustomerDal obj = new CustomerDal();

        return obj.getCustomers();
    }
    public void Add()
    {
        CustomerDal obj = new CustomerDal();
        obj.Add(this);
    }
}
```

لایه UI هم با تعریف یک Customer و فراخوانی operation های مربوط به آن، داده‌ی مد نظر خود را در قالب CustomerEntity بازیابی خواهد کرد. اگر بخواهیم عمکرد روش uniform را خلاصه کنیم باید بگوییم، در این روش دیتای رد و بدل شده‌ی مابین کلیه لایه‌ها با یک ساختار استاندارد، یعنی Entity پاس داده می‌شوند.



مزایا و معایب روش uniform

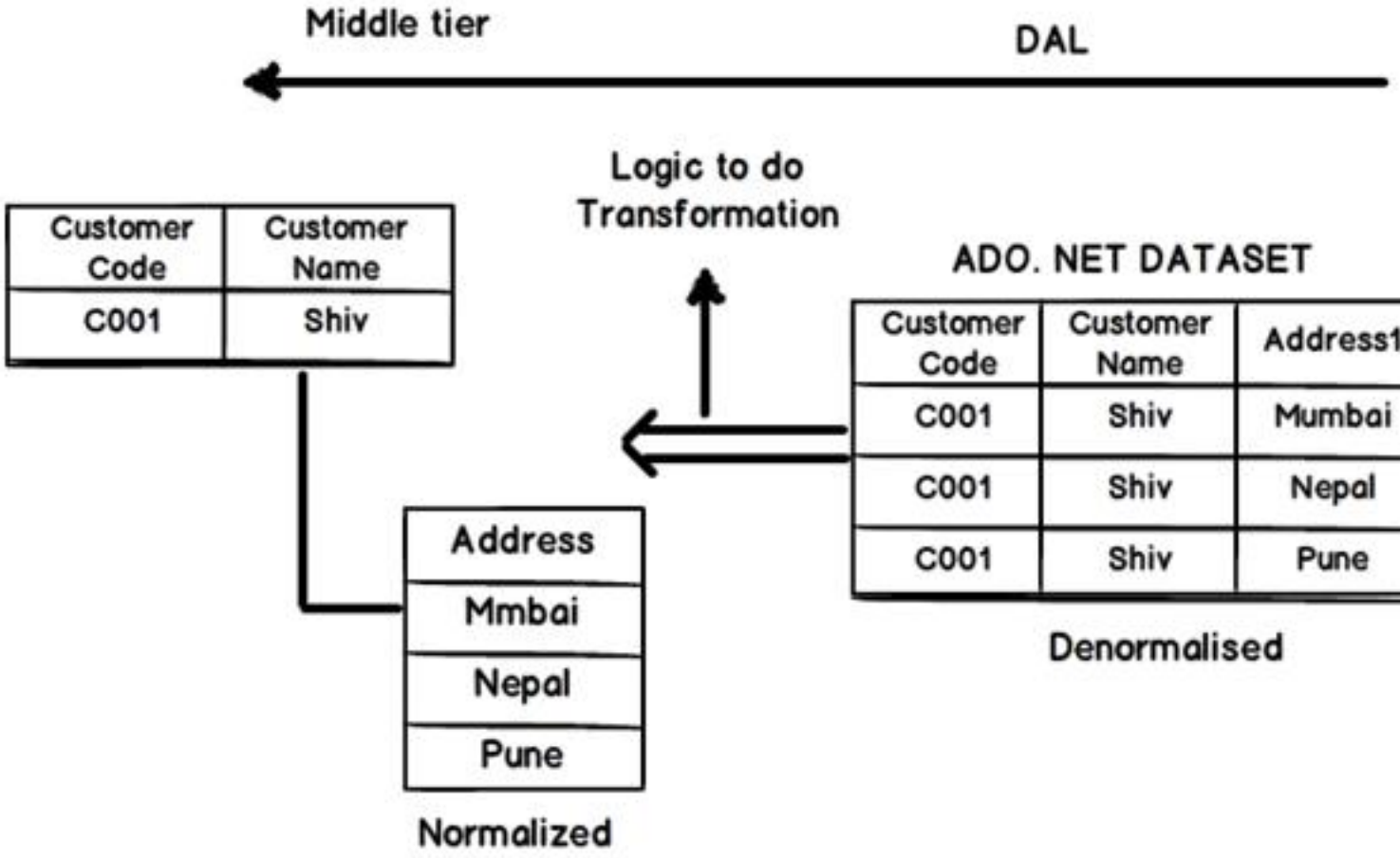
مزایا

- Strongly typed به صورت در تمامی لایه‌ها قابل دسترسی و استفاده می‌باشد.

· به دلیل اینکه از ساختار عمومی Entity استفاده می‌کند، بنابراین فقط یکبار نیاز به تبدیل داده‌ها وجود دارد. به این معنی که کافی است یک بار دیتای واکشی شده از دیتابیس را به یک ساختار Entity تبدیل کنید و در ادامه بدون هیچ تبدیل دیگری از این Entity استفاده کنید.

معایب

· تنها مشکلی که این روش دارد، مشکلی است به نام Double Loop . هنگامیکه شما در مورد کلاس‌های entity بحث می‌کنید، ساختارهای دنیای واقعی را مدل می‌کنید. حال فرض کنید شما به دلیل یکسری مسایل فنی دیتابیس خود را Optimize کرده اید. بنابراین ساختار دنیای واقعی با ساختاری که شما در نرم افزار مدل کرده‌اید متفاوت می‌باشد. بگذارید یک مثال بزنیم؛ فرض کنید که یک customer دارید، به همراه یکسری Address . همان طور که ذکر کردیم، به دلیل برخی مسایل فنی (denormalized) به صورت یک جدول در دیتا بیس ذخیره شده است. بنابراین سرعت واکشی اطلاعات بیشتر است. اما خوب اگر ما بخواهیم این ساختار را در دنیای واقعی بررسی کنیم، ممکن است با یک ساختار یک به چند مانند شکل ذیل برخورد کنیم.



بنابراین مجبوریم یکسری کد جهت این تبدیل همانند کد ذیل بنویسیم.

```
foreach (DataRow o1 in oCustomers.Tables[0].Rows)
{
    obj.Add(new CustomerEntyityAddress()); // Fills customer
    foreach (DataRow o in oAddress.Tables[0].Rows)
    {
        obj[0].Add(new AddressEntity()); // Fills address
    }
}
```

عنوان: **تبادل داده‌ها بین لایه‌ها؛ قسمت آخر**
نویسنده: سید ریوف مدرسی
تاریخ: ۱۳۹۴/۰۶/۰۳ ۲۰:۴۵
آدرس: www.dotnettips.info
گروه‌ها: ADO.NET, Design patterns, Architecture, OOP, N-Layer Architecture, Architectural Patterns

روش سوم: DTO (Data transfer objects)

در قسمت‌های قبلی دو روش از روش‌های موجود جهت تبادل داده‌ها بین لایه‌ها، ذکر گردید و علاوه بر این، مزایا و معایب هر کدام از آنها نیز ذکر شد. در این قسمت دو روش دیگر، به همراه مزایا و معایب آنها برشمرده می‌شود. لازم به ذکر است هر کدام از این روش‌ها می‌تواند با توجه به شرایط موجود و نظر طراح نرم افزار، دارای تغییراتی جهت رسیدن به یکسری اهداف و فاکتورها در نرم افزار باشد.

در این روش ما سعی می‌کنیم طراحی کلاس‌ها را به اصطلاح مسطح (flatten) کنیم تا بر مشکل double loop که در قسمت قبل بحث کردیم غلبه کنیم. در کد ذیل مشاهده می‌کنید که چگونه کلاس CusomerDTO از CustomerEntity ، مشتق می‌شود و کلاس Address را با CustomerEntity ادغام می‌کند؛ تا برای افزایش سرعت لود و نمایش داده‌ها، یک کلاس de-normalized شده ایجاد نماید.

```
public class CustomerDTO : CustomerEntity
{
    public AddressEntity _Address = new AddressEntity();
}
```

در کد ذیل می‌توانید مشاهده کنید که چگونه با استفاده از فقط یک loop یک کلاس de-normalized شده را پر می‌کنیم.

```
foreach (DataRow o1 in oCustomers.Tables[0].Rows)
{
CustomerDTO o = new CustomerDTO();
o.CustomerCode = o1[0].ToString();
o.CustomerName = o1[1].ToString();
o._Address.Address1 =  o1[2].ToString();
o._Address.Address2 = o1[3].ToString();
obj.Add(o);
}
```

UI هم به راحتی می‌تواند DTO را فراخوانی کرده و دیتا را دریافت کند.

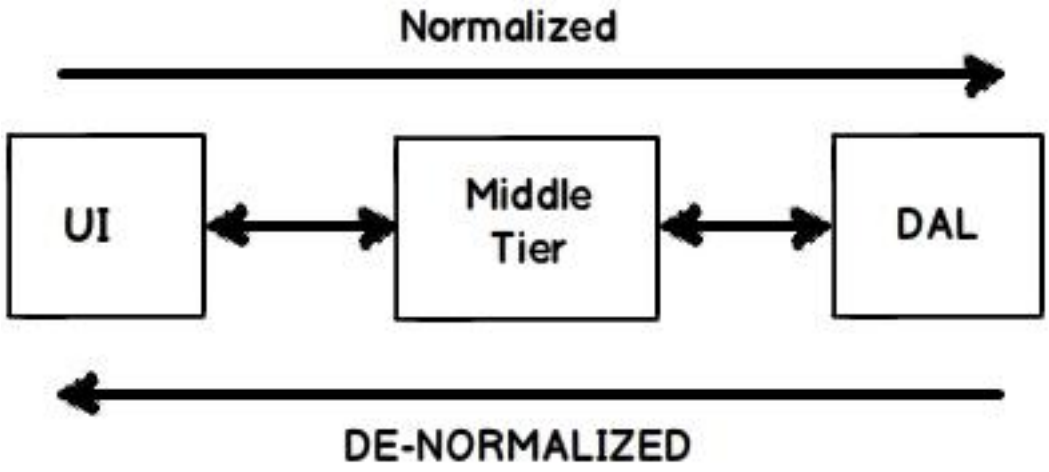
مزایا و معایب روش DTO

یکی از بزرگترین مزایای این روش سرعت زیاد در بارگذاری اطلاعات، به دلیل استفاده کردن از ساختار de-normalized می‌باشد. اما همین مسئله خود یک عیب محسوب می‌شود؛ به این دلیل که اصول شئ گرایی را نقض می‌کند.

روش چهارم: Hybrid approach (Entity + DTO)

از یک طرف کلاس‌های Entity که دنیای واقعی را مدل خواهند کرد و همچنین اصول شئ گرایی را رعایت می‌کنند و از یک طرف دیگر DTO نیز یک ساختار flatten را برای رسیدن به اهداف کارآیی دنبال خواهند کرد. خوب، به نظر می‌رسد که بهترین کار استفاده از هر دو روش و مزایای آن روش‌ها باشد.

زمانیکه سیستم، اهدافی مانند انجام اعمال CRUD را دنبال می‌کند و شما می‌خواهید مطمئن شوید که اطلاعات، دارای integrity می‌باشند و یا اینکه می‌خواهید این ساختار را مستقیما به کاربر نهایی ارائه دهید، استفاده کردن از روش (Entity) به عنوان یک روش normalized می‌تواند بهترین روش باشد. اما اگر می‌خواهید حجم بزرگی از دیتا را نمایش دهید، مانند گزارشات طولانی، بنابراین استفاده از روش DTO با توجه به اینکه یک روش de-normalized به شمار می‌رود بهترین روش می‌باشد.



کدام روش بهتر است؟

Non-uniform : این روش برای حالتی است که متدهای مربوط به data access تغییرات زیادی را تجربه نخواهند کرد. به عبارت دیگر، اگر پروژه‌ی شما در آینده دیتابیس‌های مختلفی را مبتنی بر تکنولوژی‌های متفاوت، لازم نیست پشتیبانی کند، این روش می‌تواند بهترین روش باشد.

Uniform : Entity, DTO, or hybrid : اگر امکان دارد که پروژه‌ی شما با انواع مختلف دیتابیس‌ها مانند Oracle و Postgres ارتباط برقرار کند، استفاده کردن از این روش پیشنهاد می‌شود.