عنوان مقاله: پیاده سازی معماری Domain Driven Design و تست نویسی + راهنمای گام به گام

نویسنده مقاله: تیم فنی نیک آموز

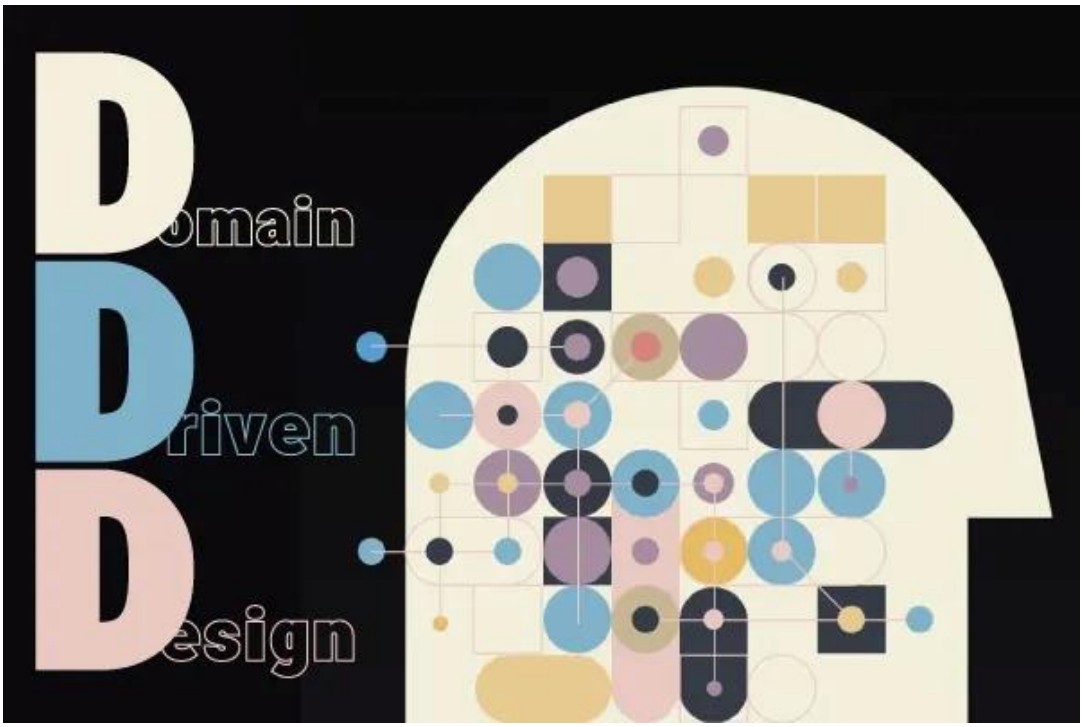
تاریخ انتشار: ۱۲ آذر ۱۴۰۲

منبع: https://nikamooz.com/implementation-of-domain-driven-design-architecture/

پیاده سازی معماری DDD یا همان طراحی دامنه محور ، نقش کلیدی در توسعه پروژه‌های نرم افزاری پیچیده دارا است. در مقاله قبلی، نحوه پیاده سازی معماری تمیز (Clean Architecture) به‌صورت گام به گام بررسی شد، حال در این مطلب که بخش دوم آن مقاله است، نحوه پیاده سازی معماری DDD آموزش داده خواهد شد.

Design Driven Domain چیست ؟

طراحی دامنه محور (DDD | Domain Driven Design) یک رویکرد در توسعه نرم افزار محسوب می‌شود که در آن، تمرکز روی درک و مدل‌سازی صحیح دامنه کسب و کار است. به واسطه پیاده سازی معماری DDD ، این موقعیت فراهم می‌شود تا ذینفعان فنی و غیرفنی بتوانند به درک مشترکی از دامنه مسئله برسند و به‌صورت مؤثر و بهبودیافته با سایر اعضای تیم مشارکت داشته باشند. با این مقدمه، در ادامه این مطلب، به نحوه پیاده سازی طراحی مبتنی بر دامنه می‌پردازیم.

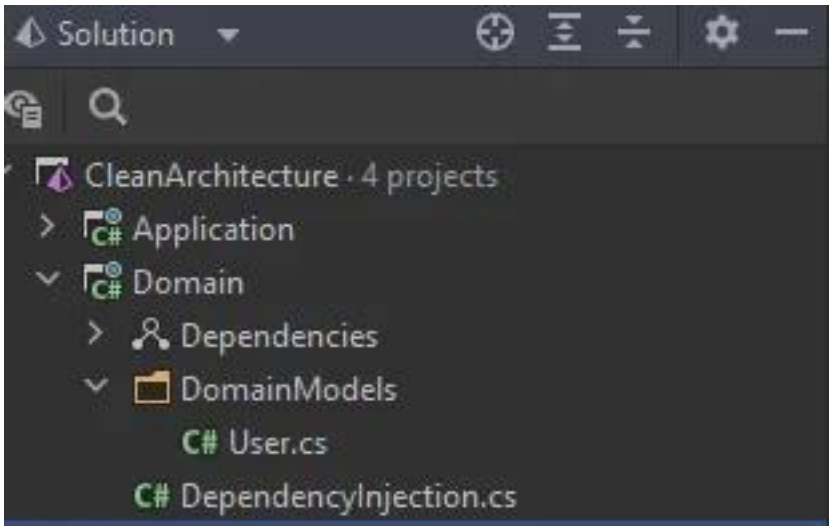


پیاده سازی معماری Domain Driven Design

پیاده سازی معماری DDD و یادگیری آن، می‌تواند به تعامل کارآمد میان اعضای تیم توسعه، بهبود نگهداری کد و راه حل‌های نرم افزاری مناسب منجر شود؛ به طوری که نیازمندی‌های مربوط به حوزه کسب و کار پاسخ داده شوند. مراحل پیاده سازی معماری Domain Driven Design به شرح زیر است:

تبدیل Anemic Model به Rich Model

به منظور شروع مسیر پیاده سازی معماری DDD ، ابتدا باید لایه Domain پروژه مورد بررسی قرار بگیرد. فرض کنید یک کلاس به نام User وجود دارد و این کلاس، در داخل پوشه‌ای به نام DomainModels (در لایه Domain) قرار گرفته است.



کلاس User مشابه زیر است:

```
public class User
{
    public string Name { get; set; }
    public string Family { get; set; }
    public string Email { get; set; }
}
```

به زبان ساده، هر کلاسی که فقط Property را شامل شود و هیچ سازنده (Constructor) و متدی (Method) نداشته باشد، Anemic Model نام دارد. اکنون قصد داریم روند پیاده سازی معماری DDD را از تبدیل یک مدل Anemic به Rich Model آغاز کنیم. Rich Model نوعی کلاسی محسوب می‌شود که در آن Behavior ها و منطق بیزینس یک مدل وجود دارد.

در Domain Driven Design ، یک شی به نام **Aggregate Root** قرار دارد که به منظور پیاده‌سازی رفتار یک مدل براساس منطق‌های بیزینسی مربوط به آن به کار می‌رود. در این‌جا، می‌توان User را به‌عنوان Aggregate Root در نظر گرفت. برای تبدیل Anemic Model به Rich Model، لازم است Setter Property های مدل User را Private کنیم. این کار به شما تضمین می‌دهد که تغییر مقادیر مربوطه، بیرون از مدل امکان‌پذیر نباشد و این عمل، فقط ازطریق متدهای موجود در داخل کلاس Aggregate قابل انجام باشد.

- **ایجاد مدل User**

حال این سؤال پیش می‌آید که چگونه می‌توان یک مدل User ایجاد و از آن استفاده کرد؟ این عمل، ازطریق دو روش زیر ممکن است.

۱. تعریف Constructor
۲. تعریف یک متد Static به نام Create

تعریف این متد استاتیک باید به نحوی باشد که تمام پارامترهای مورد نیاز ازطریق آرگومان آن دریافت و مقداردهی شوند. به منظور تسهیل کار، در این مطلب از راه اول، یعنی تعریف Constructor، استفاده خواهیم کرد.

- **افزودن منطق بیزینسی (Business Logic)**

قدم بعدی، افزودن منطق بیزینسی (Business Logic) است.

به سؤال‌های زیر توجه کنید:

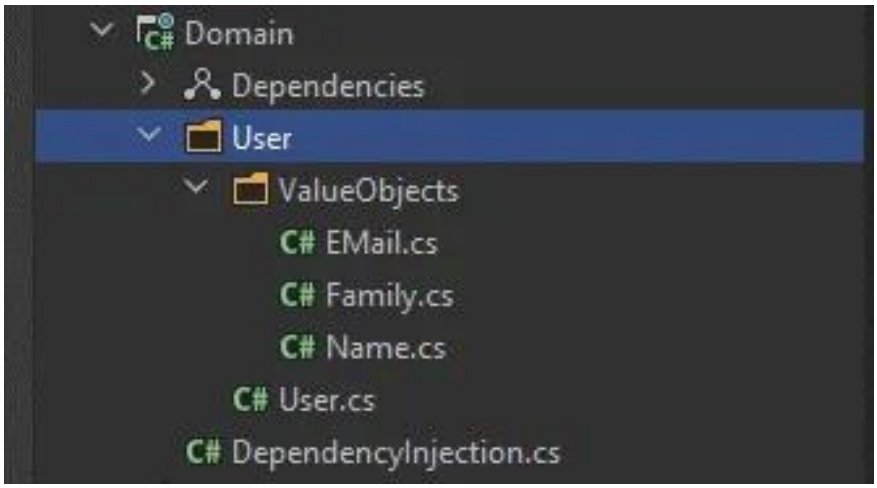
۱. آیا ایمیل واردشده از نظر ساختار صحیح است یا خیر؟
۲. آیا تعداد کاراکترهای مورد استفاده برای Name نباید از ۱۰ عدد بیشتر باشد؟
۳. آیا مقدار Family نباید شامل کلمات ممنوعه باشد؟

به پرسش‌هایی همچون موارد فوق، تحت عنوان Guard های بیزینسی اشاره می‌شود و لازم است فراخوانی و صحت‌سنجی آن‌ها در زمان ایجاد یک مدل انجام شود. برای درک بهتر، به قطعه کد زیر توجه کنید.

```
public class User
{
    public string Name { get; private set; }
    public string Family { get; private set; }
    public string Email { get; private set; }

    public User(string name, string family, string email)
    {
        if (string.IsNullOrEmpty(name) || string.IsNullOrEmpty(family) ||
string.IsNullOrEmpty(email))
        {
            throw new ArgumentException("Name, family, and email cannot be
empty.");
        }

        if (name.Length > 10)
        {
            throw new ArgumentException("Name length must not be greater
than 10.");
        }

        if (!IsValidEmail(email))
        {
            throw new ArgumentException("Invalid email format.");
        }

        Name = name;
        Family = family;
        Email = email;
    }

    private bool IsValidEmail(string email)
    {
        // منطق بیزینسی برای اعتبارسنجی ایمیل
        // در اینجا ممکن است یک الگوی اعتبارسنجی ساده استفاده شود
        return Regex.IsMatch(email, @"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-
]+\.[a-zA-Z]{2,}$");
    }
}
```

تا این بخش از پیاده سازی معماری DDD ، یک مدل ساده را به نوع Rich آن تبدیل کرده‌ایم. اما هنوز یک ایراد بر آن وارد است. به مرور زمان، با افزایش تعداد Guard های بیزینس، رفع اشکالات مربوط به Aggregate Root و تغییر دادن یا اضافه کردن کد به آن، با چالش همراه خواهد بود. به منظور رسیدگی به این مسئله، لازم است بیزینس مربوط به هر Property در داخل یک کلاس جدا بررسی شود.

برای بهبود ساختار پروژه، اسم پوشه DomainModes رو به User تغییر دهید و در داخل آن یک پوشه به نام ValueObjects ایجاد کنید. اکنون، مشابه تصویر زیر، سه کلاس به نام‌های Name, Family و Email داخل پوشه ValueObjects بسازید.



به این ترتیب، هر پوشه‌ای که مربوط به دامین پروژه (User) باشد، دارای یک AggregateRoot و شامل تعدادی ValueObjects خواهد بود. برای اینکه در کدها مشخص شود کدام کلاس‌ها Value Object و کدام Aggregate Root هستند، لازم است تعدادی کلاس Generic تعریف کنیم؛ به طوری که تمام Aggregate Root ها از کلاس جنریک AggregateRoot و تمام Value Object ها از کلاس جنریک ValueObject ارث بری کنند.

پیاده سازی کلاس های جنریک (Generic Classes)

پیش از پیاده‌سازی کلاس‌های جنریک، این سؤال مطرح می‌شود که محل پیاده‌سازی آن‌ها در کدام لایه خواهد بود؟

برای پیاده‌سازی کلاس‌های جنریک، راه حل‌های پایینی پیش روی شما قرار دارند:

- **روش اول:** شما می‌توانید یک پروژه جداگانه به نام framework ایجاد کرده و پیاده‌سازی‌های جنریک را در آن قرار دهید.
- **روش دوم:** در این رویکرد، باید یک پروژه دیگر به نام GenericDomain ایجاد کرده و مفاهیم مربوط به DDD را در آن پیاده‌سازی می‌کنید.

با توجه به شرایط پروژه و نیازمندی‌های آن، می‌توان راه حل مناسب را انتخاب کرد.

پیاده سازی کلاس AggregateRoot ازطریق قطعه زیر انجام می‌شود.

```
public abstract class AggregateRoot<TKey>
{
    public TKey Id { get; set; }
    public DateTime CreateAt { get; set; }
    public DateTime? ModifiedAt { get; set; }
}
```

کلاس AggregateRoot از نوع Abstract است و فیلد TKey جنس شی Id را مشخص می‌کند. همچنین، زمان ایجاد و ویرایش هر Aggregate Root، در property های CreateAt و ModifiedAt ذخیره و نگهداری می‌شوند.

پیاده‌سازی کلاس ValueObject به‌صورت زیر انجام می‌شود.

```
public abstract class ValueObject
{
    protected static void CheckRule(IBusinessRule rule)
    {
        if (!rule.HasValidRule())
        {
            throw new BusinessRuleValidationException(rule);
        }
    }
}
```

کلاس ValueObject از نوع Abstract است و فقط دارای یک متد خواهد بود. این متد از نوع Protected یا همان محافظت شده است. این یعنی، متد مذکور فقط در کلاس‌هایی قابل فراخوانی هستند که از کلاس ValueObject ارث بری می‌کنند.

متد CheckRule به منظور بررسی معتبربودن منطق بیزینسی به‌کار می‌رود؛ به طوری که یک شی از نوع IBusinessRule دریافت می‌کند و اگر مقدار متد HasValidRule صحیح نباشد، یک Custom Exception رخ می‌دهد. در چنین شرایطی، مقدار rule به‌عنوان ورودی دریافت می‌شود.

قطعه کد زیر مربوط به اینترفیس IBusinessRule است.

```
public interface IBusinessRule
{
    bool HasValidRule();
    string Message { get; }
}
```

اینترفیس IBusinessRule متشکل از یک متد و یک Property است. کلاسی که از این اینترفیس ارث بری می‌کند، باید هم متد و هم Property این اینترفیس را پیاده‌سازی کند؛ به نحوی که اگر منطق نوشته‌شده در متد HasValidRule نقض شد، پیام در نظر گرفته شده برای Message مورد استفاده قرار بگیرد.

به منظور درک نحوه استفاده متغیر Message، قطعه کد مربوط به کلاس BusinessRuleValidationException را مشاهده کنید:

```
public class BusinessRuleValidationException : Exception
{
    public BusinessRuleValidationException(IBusinessRule brokenRule) :
base(brokenRule.Message)
    {

    }
}
```

Class فوق از کلاس Exception ارث بری می‌کند، سپس در Constructor آن یک شی، از جنس brokenRule دریافت می‌شود و مقدار متغیر Message به کلاس base مربوطه پاس داده خواهد شد. به این ترتیب، هر زمان که منطق بیزینسی نقض شود، پیغام خطای مناسبی به‌صورت Exception نمایش داده می‌شود.

اکنون قصد داریم به بررسی نحوه استفاده از کلاس‌های جنریک تعریف‌شده بپردازیم.

در این مرحله از پیاده سازی معماری DDD ، لازم است کلاس User از AggregateRoot ارث بری شود و Guid به‌عنوان مقدار TKey در نظر گرفته شود.

```
public class User : AggregateRoot<Guid>
```

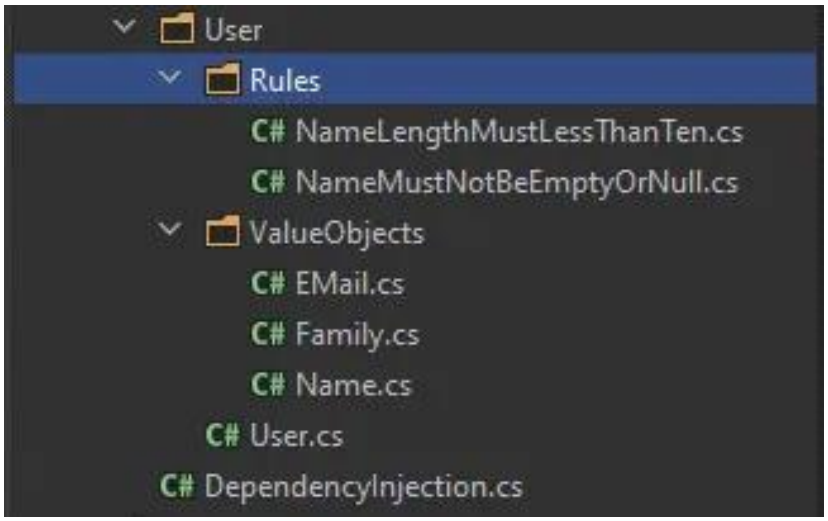
حال، باید کلاس‌های Family, Name و Email از کلاس جنریک ValueObject ارث بری شوند.

نتیجه به‌صورت زیر خواهد بود:

```
public class Name : ValueObject
{
    private readonly string _name;
    public string Value => _name;

    public Name(string name)
    {
        CheckRule(new NameMustNotBeEmptyOrNull(name));
        CheckRule(new NameLengthMustLessThanTen(name));
        _name = name;
    }
}
```

کلاس Name از دو Property، یعنی Private و Public تشکیل شده است. مقدار متغیر Private همیشه ازطریق Constructor یا همان سازنده کلاس تعیین می‌شود و راه دیگری برای تغییر آن وجود ندارد. ضمن اینکه متغیر Value فقط برای ارائه مقدار کلاس Name مورد استفاده قرار می‌گیرد .به این ترتیب، می‌توان تضمین کرد این متغیر تنها در شرایطی مقدار خواهد داشت که پس از فراخوانی، هر دو متد CheckRule منجر به Exception نشوند. سایر کلاس‌ها، نظیر Email و Family نیز به همین شکل تغییر خواهند کرد. کلاس‌های Name ,Family و Email اصطلاحاً Strong Type نامگذاری می‌شوند.

در کلاس Name دو شی جدید، به نام‌های NameMustNotBeEmptyOrNull و NameLengthMustLessThanTen وجود دارند. بدون مشاهده پیاده‌سازی‌های این دو کلاس، می‌توان کارایی هر یک از آن‌ها را حدس زد. این موضوع یکی از اصول اساسی کدنویسی تمیز محسوب می‌شود که در پیاده سازی معماری DDD به‌کار رفته است. دو کلاس NameMustNotBeEmptyOrNull و NameLengthMustLessThanTen در پوشه‌ای به نام Rules (داخل پوشه User) ایجاد شده‌اند.



مشابه تصویر فوق، هر آن چه که از مدل User نیاز دارید، به‌صورت دسته‌بندی شده و مشخص در کنار هم قرار دارند. شما حتی می‌توانید Exception هایی که مربوط به کاربر هستند را در داخل پوشه Exceptions (زیرمجموعه پوشه User) قرار دهید. با توجه به اینکه در این آموزش، یک نوع Exception داریم و به‌صورت عمومی استفاده می‌شود، این مورد در داخل پوشه Exception (در root پروژه Domain) قرار داده شده است.

پیاده‌سازی کلاس‌های NameLengthMustLessThanTen و ValueMustNotBeEmptyOrNull ازطریق کد زیر انجام خواهد شد:

```
public class ValueMustNotBeEmptyOrNull : IBusinessRule
{
    private readonly string _value;
    public ValueMustNotBeEmptyOrNull(string value)
    {
        _value = value;
    }
    public bool HasValidRule()
    {
        var isValid = !string.IsNullOrEmpty(_value);
        return isValid;
    }

    public string Message => $"The value of {_value} must not be null or
empty.";
}
```

کلاس ValueMustNotBeEmptyOrNull از اینترفیس IBusinessRule ارث بری کرده و باید متد HasValidRule و Message را پیاده‌سازی کند. متغیر name ازطریق Constructor یا سازنده کلاس دریافت می‌شود و سپس، در داخل متد HasValidRule بررسی خواهد شد. اگر مقدار name برابر با null یا empty باشد، مقدار isValid برابر با false می‌شود. در چنین شرایطی، پیغام خطای در نظر گرفته شده برای Message، استفاده خواهد شد.

```
public class NameLengthMustLessThanTen : IBusinessRule
{
    private readonly string _name;

    public NameLengthMustLessThanTen(string name)
    {
        _name = name;
    }

    public bool HasValidRule()
    {
        var isValid = _name.Length >= 10;
        return isValid;
    }

    public string Message => $"The length of the {_name} must not be
greater that 10.";
}
```

رفتاری که برای ValueMustNotBeEmptyOrNull شرح داده شد، برای NameLengthMustLessThanTen نیز صادق است؛ اما isValid در شرایطی false خواهد شد که طول رشته کاراکتر name بیشتر از ۱۰ عدد باشد. در این صورت، پیغام خطای مربوط به شرح نقض بیزینس، نمایش داده خواهد شد.

ریفکتورینگ کلاس های ایجاد شده

در این گام از پیاده سازی معماری DDD ، باید Refactoring کلاس User را با استفاده از کلاس‌های ساخته شده انجام دهیم. خروجی این عمل، به شکل زیر خواهد بود:

```
public class User : AggregateRoot<Guid>
{
    public Name Name { get; private set; }
    public Family Family { get; private set; }
    public EMail EMail { get; private set; }

    public User(string name, string family, string email)
    {
        Name = new Name(name);
        Family = new Family(family);
        EMail = new EMail(email);
    }
}
```

تمامی رفتارها و منطق بیزینسی که قرار بود برای ایجاد User لحاظ شود، در قالب چند کلاس Encapsulate شده‌اند و در نهایت، یک کلاس تمیز و خوانا حاصل شد.

بررسی Business Logic

در این مرحله با کمک Aggregate Root ایجاد شده، یک User در لایه Application ایجاد می‌کنیم؛ به گونه‌ای که تمامی منطق‌های بیزینسی مورد نظر چک شوند. برای درک بهتر، به قطعه کد پایین توجه کنید.

```
public class CreateUserCommandHandler : IRequestHandler<CreateUserCommand,
bool>
{
    public Task<bool> Handle(CreateUserCommand request, CancellationToken
cancellationToken)
    {
        var user = new User(request.Name, request.Family, request.Email);
        return Task.FromResult(true);
    }
}
```

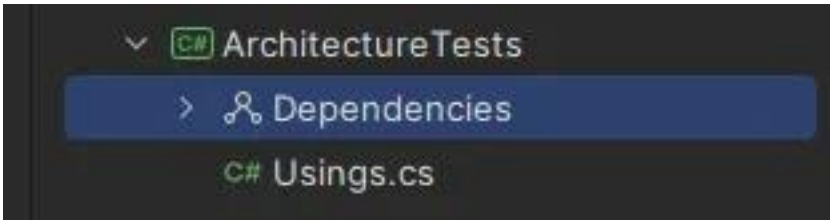
در کلاس CreateUserCommandHandler (در لایه Application)، یک متد به نام Handler وجود داشت که پیش از این فقط مقدار true را برمی‌گرداند. اکنون در داخل این متد، باید یک نمونه از کلاس User ایجاد شود و مقادیر نام، نام خانوادگی و ایمیل یک کاربر دریافت و به آن ارسال شود. در داخل کلاس User، مقادیر از دید منطق بیزینسی بررسی خواهند شد و اگر مشکلی وجود نداشته باشه، درنهایت مقدار true بازگردانده می‌شود. در غیر این صورت، یک Exception با پیام مشخص نمایش داده خواهد شد.

این یک پیاده‌سازی ساده از Domain Model در لایه Domain و نحوه استفاده از آن در لایه Application بود. البته جزئیات بسیار زیادی را می‌توان به این مدل‌ها اضافه کرد. به‌عنوان نمونه، ممکن است یک سیستم از چندین Domain Model تشکیل شده باشد، به گونه‌ای که بتوان دو یا چند یک از آن‌ها را به‌عنوان یک Aggregate Root واحد تعریف کرد. مثلاً سیستم‌های مربوط به پرداخت، می‌توانند به ازای هر Payment (مانند پرداخت با درگاه بانکی، استفاده از کیف پول)، چند نوع Transaction داشته باشند و برای تضمین معتبر بودن آن‌ها از حیث رفتار و قوانین بیزینسی، باید همه آن‌ها را در قالب یک Aggregate Root تعریف کرد.

نوشتن تست برای بررسی معماری DDD

در این گام از مطلب پیاده سازی معماری DDD ، نحوه برقراری قوانین طراحی و معماری در پروژه بررسی می‌شود. به‌عنوان مثال، می‌توانید محدودیت‌هایی برای ارتباط بین لایه‌های مختلف قرار بدهید یا الگوهای نامگذاری مشخصی را برای برخی از کلاس‌ها در نظر بگیرید؛ به طوری که سایر توسعه‌دهندگان نتوانند از آن تخطی کنند.

ابتدا یک پروژه جدید به نام Architecture Test و از نوع Unit Test ایجاد کنید. در این مقاله، ما از xUnit برای تست استفاده خواهیم کرد.



برای شروع، لازم هست که Nuget Package را ازطریق دستورات زیر نصب کنید. Nuget Package ابزاری است به منظور پیاده‌سازی موارد مذکور استفاده می‌کنیم.

```
dotnet add package NetArchTest.Rules --version 1.3.2
dotnet add package FluentAssertions --version 6.12.0
```

پیش از استفاده از این پکیج، لازم است یک تغییر جزئی در کلاس‌های Domain و Value Object اعمال کنید. این یعنی، باید تمام کلاس‌های Family, Name, Email, User را به‌صورت sealed قرار دهیم تا امکان ارث بری از این کلاس‌ها در سایر کلاس‌ها وجود نداشته باشد.

```
public sealed class User : AggregateRoot<Guid>
public sealed class Name : ValueObject
public sealed class Family : ValueObject
public sealed class EMail : ValueObject
```

حالا با کمک NetArchTest بررسی خواهیم کرد که آیا تمام کلاس‌های نامبرده sealed هستند یا خیر. برای اینکار، ابتدا یک پوشه به نام Domain در داخل پروژه تست بسازید و سپس در داخل آن، یک کلاس تحت عنوان DomainTests ایجاد کنید.

کد زیر نحوه پیاده‌سازی این موضوع است:

```
private static readonly Assembly DomainAssembly =
typeof(DependencyInjection).Assembly;

[Fact]
public void ValueObjects_Should_BeSealed()
{
    var result = Types.InAssembly(DomainAssembly)
        .That()
        .AreClasses()
        .And()
        .Inherit(typeof(ValueObject))
        .Should()
        .BeSealed()
        .GetResult();

    result.IsSuccessful.Should().BeTrue();
}
```

در قدم اول، لازم است از Assembly لایه Domain برای پیدا کردن کلاس‌ها استفاده شود که دستور زیر از کد بالا، همین کار را انجام می‌دهد.

```
private static readonly Assembly DomainAssembly =
typeof(DependencyInjection).Assembly;
```

مهم نیست از کدام کلاس به جای DependencyInjection استفاده کنیم. همین که Assembly آن را دریافت کنیم، کافی است. این یعنی، می‌توانیم به اشیای لایه Domain دسترسی داشته باشیم.

در قدم بعد، گفته خواهد شد که داخل Assembly، لازم است هر شی که کلاس محسوب می‌شود و از ValueObject ارث بری کرده باشد، sealed باشد. نتیجه حاصل در داخل متغیر result ذخیره می‌شود و در انتها بررسی می‌کنیم آیا نتیجه تست درست بوده یا خیر.

```
result.IsSuccessful.Should().BeTrue();
```

در تکه کد بالا، عبارت‌های Should و BeTrue از پکیج FluentAssertation هستند. به واسطه این عبارات، کار تست نویسی به زبان گفتاری نزدیک‌تر می‌شود.

مشابه ValueObject، حال باید بررسی شود که آیا AggregateRoot ها نیز sealed هستند یا خیر. برای درک بهتر، به کد زیر توجه کنید.

```
[Fact]
public void AggregateRoots_Should_BeSealed()
{
    var result = Types.InAssembly(DomainAssembly)
        .That()
        .AreClasses()
        .And()
        .Inherit(typeof(AggregateRoot<>))
        .Should()
        .BeSealed()
        .GetResult();

    result.IsSuccessful.Should().BeTrue();
}
```

تست نویسی بعدی درمورد رعایت اصول نامگذاری است. به‌عنوان مثال، قصد داریم در انتهای نام همه کلاس‌های Handler، عبارت Handler وجود داشته باشد. در ادامه، به پیاده‌سازی این عمل می‌پردازیم.

هندلرها در لایه Application قرار دارند؛ به همین دلیل، یک پوشه به اسم Application ایجاد کرده و سپس در داخل آن یک کلاس به اسم ApplicationTests بسازید. توجه کنید که ابتدا باید Assembly این لایه به‌دست آورده شود:

```
private static readonly Assembly ApplicationAssembly = typeof(Application.DependencyInjection).Assembly;
```

حال باید بررسی شود که آیا تمامی کلاس‌هایی که از اینترفیس IRequestHandler ارث بری می‌کنند، در انتهای نامشان عبارت Handler را دارا هستند. این عمل توسط قطعه کد زیر انجام می‌شود:

```
[Fact]
public void Handler_Should_HaveHandlerAtTheEndOfItsName()
{
    var result = Types.InAssembly(ApplicationAssembly)
        .That()
        .ImplementInterface(typeof(IRequestHandler<,>))
        .Should()
        .HaveNameEndingWith("Handler")
        .GetResult();

    result.IsSuccessful.Should().BeTrue();
}
```

تست نویسی پایانی برای این پیاده سازی معماری DDD ، بررسی ارتباط بین لایه‌های مختلف پروژه است. برای این کار، یک کلاس به نام LayerTests ایجاد کنید. می‌خواهیم به واسطه تست نویسی، تضمین شود که لایه Domain هیچ وابستگی خاصی به لایه Application نخواهد داشت. این عمل، ازطریق پیاده‌سازی متد زیر قابل انجام است:

```
private static readonly Assembly DomainAssembly =
typeof(DependencyInjection).Assembly;

[Fact]
public void DomainLayer_ShouldNot_HaveDependencyOnApplicationLayer()
{
    var result = Types.InAssembly(DomainAssembly)
        .Should()
        .NotHaveDependencyOn("Application")
        .GetResult();

    result.IsSuccessful.Should().BeTrue();
}
```

متد NotHaveDependencyOn به جستجوی هرگونه وابستگی به Application و هر آن چیزی که در آن وجود دارد بین اشیای مختلف موجود در Assembly می‌پردازد. در صورتی که موردی توسط این متد پیدا نشود، مقدار result برابر با true خواهد شد. با کمک این روش، می‌توان وابستگی به سایر لایه‌ها را نیز بررسی کرد.

شما می‌توانید فایل‌های مربوط به این پروژه راه اندازی معماری تمیز و پیاده‌سازی DDD را از **اینجا** دانلود کنید.

سخن آخر: پیاده سازی Domain Driven Design

پیاده سازی معماری DDD و استفاده از معماری تمیز (Clean Architecture)، هردو با تأکید بر جداسازی دغدغه‌های نرم‌افزاری سیستم ارائه شده‌اند تا به کمک آن‌ها، قابلیت نگهداری و ماژولاریتی Codebase بهبود یابد. در حقیقت، این رویکردها با ایزوله‌سازی وابستگی‌ها و Unit Testing در لایه‌های مختلف، قابلیت تست‌پذیری را تقویت می‌کنند. در این مطلب، پیاده سازی معماری DDD به‌صورت مرحله‌به‌مرحله و همراه با کدنویسی مورد بررسی قرار داده شد. علاوه‌بر این، شما می‌توانید با شناخت انواع معماری توسعه نرم‌افزار، ازجمله معماری مونولیتیک و میکروسرویس، مهارت خود را در این حوزه گسترش دهید.