معماری نرم افزار چیست؟

انتخاب یک ساخاتار کلی برای پیاده سازی یک پروژه نرم افزاری بر مبنای مجموعه ای از نیازهای کاربری و تجاری یکی سیستم نرم افزاری است تا هم بتوان کاربردهای مورد نظر را پیاده سازی کرد و هم بتوان کیفیت نرم افزار تولید آن و نگهداری آن را نیز بهینه کرد و سرعت بخشید.

ساختار باید بر اساس نیاز کسب و کار ما باشد و هر کسی که معماری را ببیند بتواند بفهمد هدف معماری چیست و مربوط به چه نوع پروژه ای است.

معماری باید کاربرد نرم فزار را نمایش دهد همانند نقشه یک ساختمان.

برای معماری نباید framework , tools مهم باشد بلکه useCase مهم است.

معماری باید ایپلیکیشن را خوب ساپورت کند و از ظاهرش مشخص باشد که مثلا برای اپلیکیشن حسابداری است.

معمار خوب کسی است که :

استفاده از ابزار ها و فریم ورک ها را به تاخیر بندازد با استفاده از abstraction , dependency inversion

هدف:

هدف اصلی حمایت از چرخه حیاط سیتم است معماری خوب باعث می شود درک، نگداری و استقرار سیستم آسان باشد. هدف نهایی به حداقل رساندن هزینه، افزایش طول عمرسیستم و به حداکثر رساندن بهره وری و برنامه نویسی است.

2 نوع از معروف ترین معماری ها: data centric - domain centric

Data centric

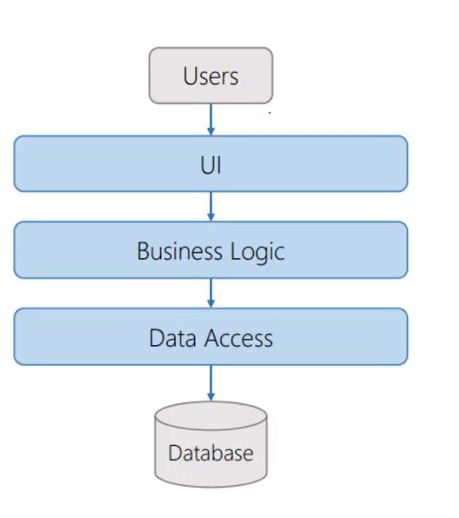
هسته اصلی نرم افزار DataBase است که خود اصول معماری نرم افزار را زیر سوال می برد اما برای حالت های که سریع می خواهیم اپلیکیشن را توسعه دهیم مناسب است اما خیلی سخت و کارایی پایین.

مزایا :

توسعه سریع

معایب :

دیتابیس مرکز نرم افزار است 0 انتخاب اشتباه مرکز ، testable نبودن ، پخش بودن قوانین کسب و کار

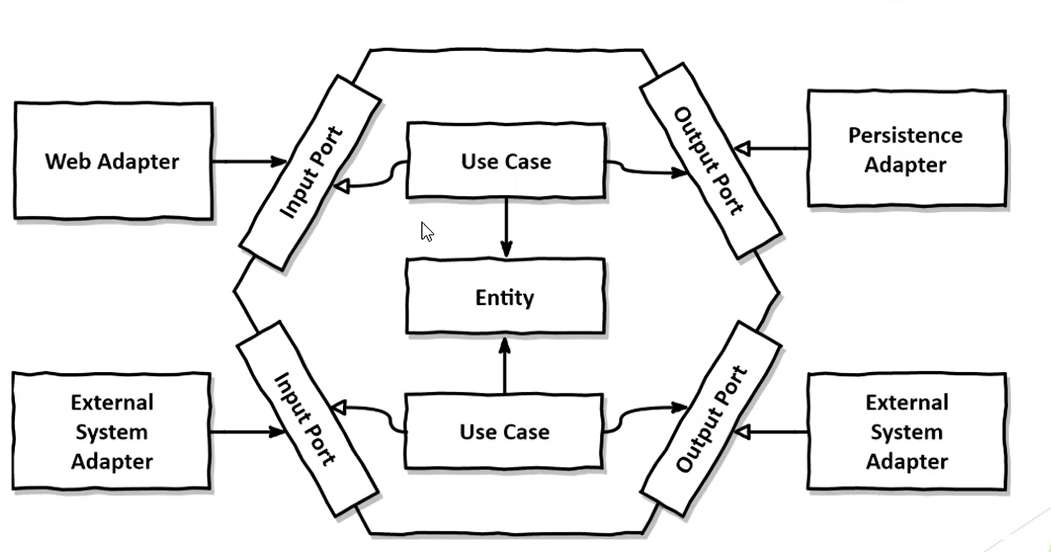


Domain centric

تمام قوانین کسب و کار و تمام موارد مربوط به کسب و کار در مرکز است و کاری با db و framework نداریم و تمامی پیش نیاز های نرم فزار را با توجه به امکانات زبان برنامه نویسی پیاده سازی کرده که سبب تست نویسی راحت تر و توسعه پذیری بیشتر می شود در ادامه مثال ها :

Hexagon

خیلی وقت است استفاده نمی شود



Onion

هسته مرکزی ما domain است و دارای یک applicationCore و قسمت های بیرونی داریم.

DomainModel

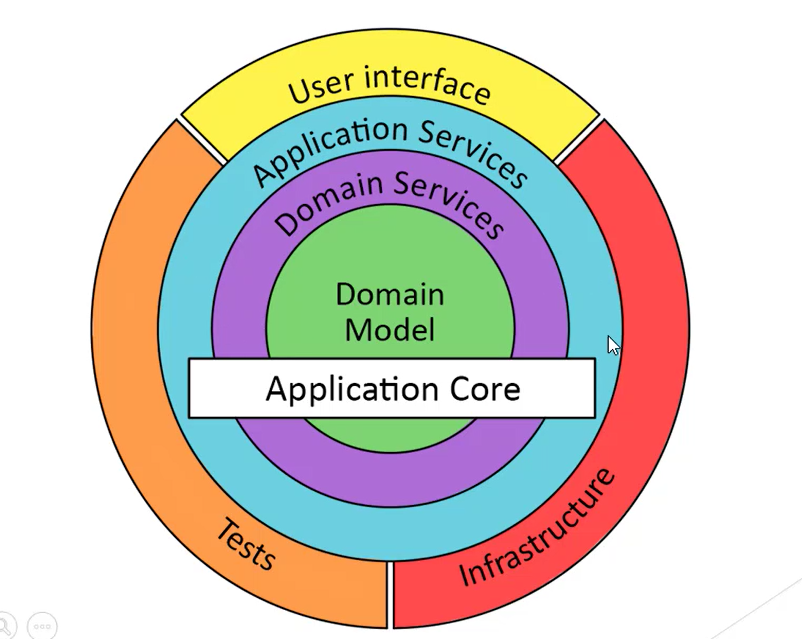
شامل entity ها و قوانین کسب و کار ما است

DomainService

سرویس های مربوط به entity ها و یک سری قوانین که در آن ها پیاده سازی شده

ApplicationService

useCase ها را شامل می شود مثلا در سیستم فروشگاهی یک useCase برای خرید مشتری داریم یک فانکشن که محصول را کم می کند به کاربر اضافه می کند و فرآیند را انجام می دهد.



Clean architecture

هدف این است که Core ما domain باشد نه database و از جزئیات دور باشیم تا زمانی که به آن نیاز داشته باشیم.

وابستگی ها به صورت لایه ای است هر لایه به قبلی خود وابسته است

Entites

در هسته مرکزی است و به هیچ چیزی وابسته نیست که قوانین کسب و کار است.

مثلا در ثبت نام کاربر شماره تکراری نیست این قانون کسب و کار است که در Entity پیاده سازی می شود.

UseCase

قوانین کسب و کار است

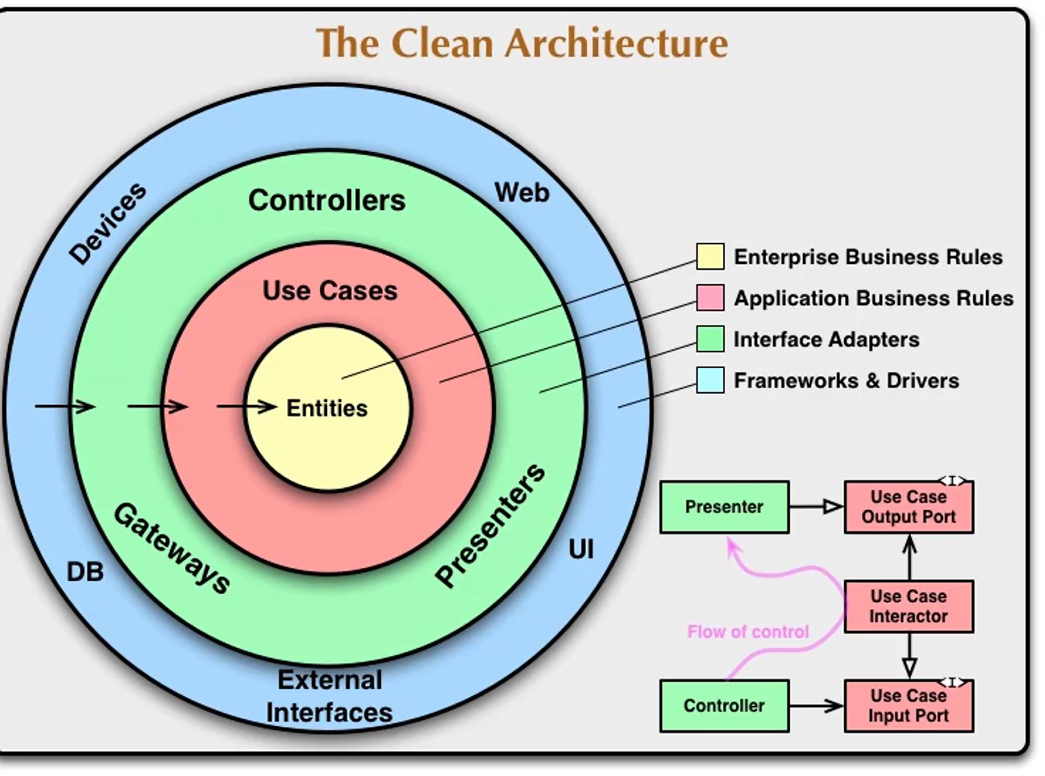
Interface adapter

مپ کردن دیتا برای ui و مپ کردن ui برای useCase ها را انجام می دهد کار input و output را انجام می دهد

لایه آخر

شامل framework ها و driver ها است

نکته: در معماری لزوما قرار نیست تمای لایه ها پیاده سازی شوند بلکه نسب به کار خود لایه ها را پیاده سازی می کنیم.



پیاده سازی :

Entites => Domin که شامل موجودیت و بیزنس رول و عدم وابستگی به هیچ چیزی غیر زبان برنامه نویسی pure باشد حتی attribute های مربوط به ef

Entity ها باید به صورت richModel باشد تا بتوان bussnes rule ها را در آن قرار داد.

در مقابل richModel نوع Anemeic قرار دارد

در انمیک قوانین را بعدا در سرویس ها اضافه می کنیم اما در richModel هر شرطی که بخواهیم داشته باشیم باید روی خود entity باشد مثلا شرط های edit باید روی خود entity باشند.

هر entity در یک فولدر قرار می گیرد.

**نکته :**

برای این که به صورت anemic موجودیت ها را ایجاد نکنیم باید تمام property های مربوط به entity به صورت private set باشد یعنی فقط توسط کلاس خود مثلا Product قابل تغییر هستند.

به همین دلیل زمانی که یک object از product بسازیم و مقادیر prop ها خالی نباشند باید در ctor تمامی prop ها را مقدار دهی کنیم.

در حالت richDomain دیگر کاربر دستبش باز نیست که هر جا دلش خواست کاری روی entity انجام دهد.

**نکته:**

ممکن است بعضی از قوانین ما مربوط به db باشد در این حالت نیز نمیاییم که db را در domain خود depend کنیم در این حالت از یک interface استفاده می کنیم به نام DomainService که در لایه خارجی implement می شود.

الگوی repository

کارش تبدیل domain model یا بیزنس مدل را به data Model تبدیل کند چیزی که در rich model ها است ربطی به چیزی که در db قرار ذخیره بشه با چه مدلی و چه ویژگی های ندارد بلکه با repository این را مشخص می کنیم.

جای که repository پیاده سازی می شود باید کاری کند که موجودیت قابلیت ذخیره در Db را داشته باشد.

نکته : یکی از راه های پوشه بندی این است که تمامی چیز های مربوط به یک entity را در یک پوشه قرار دهیم.

نکته:

UseCasek در واقع همان سرویس های ما است و چیزی که در لایه سرویس داریم باید با لایه Domain فرق کنه و باید dto باشد که به مدل domain تبدیلش می کنیم. یعنی چیزی که ui میده را مپ می کند به چیزی که useCase می خواهد و برعکسش را نیز انجام می دهد.

نکته: تا لایه ی اپلیکیشن جزو core ما حساب می شود و کاری با infrastrucre نداریم و لایه بعدی ما مسئول implement کردن است استفاده از frame work ها است.

نکته: کار لایه application در واقع orchestration است یعنی مواردی که باید پشت هم انجام شوند را انجام می دهد و کار دیگه ای انجام نمی دهد. یعنی همه کار ها قبلا در domain انجام شده اند و وظیفه سرویس این است آن ها صدا بزند و اگر کار infrastrucre داشت انجام می دهد مثل دریافت product انجام edit و save کردن آن را پشت هم انجام میدهد به این کار orchestration می گویند.

Infrastructure

لایه ی است که وظیفه آن برقراری ارتباط با db است. در این لایه interface های مربوط به لایه های پایینی را پیاده سازی می کنیم مثل repositoy ها یا لازم است با سرویس استعلام شماره کارت طرف مقابل استفاده شود پیاده سازی آن در این سرویس است و نصب پکیج در این لایه منعی ندارد.

نکته : static ها از بین نمی روند و در مموری می مانند.

Contract

لایه ای که می توان ایجاد کرد یکسری از interface های غیر مربوط را یا تمامی interface ها را در آن جا قرار دهیم.

Dpendency inversion

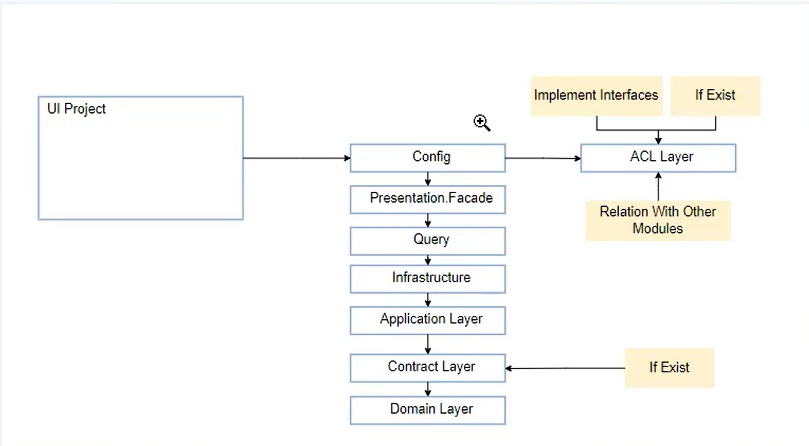
یک سرویسی مثل ISendSmsService داریم که فقط abstranction آن را تعریف می کنیم و در جا های دیگر آن را پیاده سازی می کنیم. مثلا در لایه Contract و پیاده آن را در لایه Infrastructure انجام دهیم.

Presentation

این لایه با کاربر در تعامل است در واقع front کار است. کارش صدا زدن متد های اپلیکشن و انجام دادن useCase ها است. در این لایه از dto استفاده می شود نه از خود entity

bootstrapper

در این لایه مشخص می کنیم که interface های ما مربوط به کدام پیاده سازی هستند. به این لایه endpoint نیز می گویند



Domain Drive Design

یک سری اصول و الگو ها است که با کمک آن می توانیم خیلی خوب domain خود را شناخته و درک کنیم و پیاده سازی کنیم.

Domain

هر حوزه ی تخصصی را domain می گویند. حوزه فعالیت یا دامنه ی است که در آن کار می کنیم مثلا در شرکت مالی domain ما هر چیزی مربوط به حسابداری است.

DomainModel

فرمی از دانش که به صورت آگاهانه و انتخابی ساده شده است.

مجموعه انتزاعی از دانش برای حل مشکلی است که با آن سر و کار داریم.

برای شناسایی domain باید با بیزنس من bussens expert های مربوط به آن حوزه ارتباط داشته تا مدل سازی درستی ایجاد شود.

با استفاده از اطلاعات راجب بیزنس می توانیم یک domain بزرگ را به چندین sub domain تبدیل کرده و پیاده سازی ساده تر داریم و می توانیم به صورت ماژولار پیاده سازی را انجام دهیم.

2 الگوی ارائه شده در DDD => technical pattern – strategy pattern

**Technical**

کمک می کند که بتوانیم domain model را پیاده سازی کنیم و طوری کد را بنویسیم که نرم افزار ما شسته رفته باشد و به خوبی پیاده سازی شود

**اجزا technical**

Value object

Aggregates

Entities

Domain events

Domain service

**Strategy pattern**

اصول و الگوی که به ما کمک می کند که domain را درک کرده و بهتر آن را بسازیم و ارتباط بهتری با بیزنس اکسپرت ها داشته باشیم

بررسی اجزای الگوی technical

Value object

دارای ویژگی های زیر است

Identity-less نباید دارای شناسه یا id باشد و بر اساس مقدار prop ها مقایسه می شوند.

Attribute base equality بر اساس prop ها مساوی بودنشان بررسی می شود

Behavior rich باید مانند domain ها rich مدل باشند.

Immutable نباید مقدار آن تغییر کند بلکه باید object جدید ساخته شود و آن مقدار تغییر کند.

Combinable باید با + بتوان 2 تای آن ها را جمع کرد

Self validation نباید مقدار اشتباه در آن قرار گیرد خودش Validation کند

Testable بعد پیاده سازی تمامی موارد این مورد نیز اضافه می شود.

ایجاد value object با struct و class امکان پذیر است.

تشخیص ساخت:

زمانی که یک سری validation داریم یا چند جا تکرار می شود یا مقادیر مشابه داریم validation آن ها کنار هم باشند یا می خواهیم خوانایی کد ما افزایش یابد.

باید در کنار entity قرار گیرد و state آن entity را تکمیل می کند.

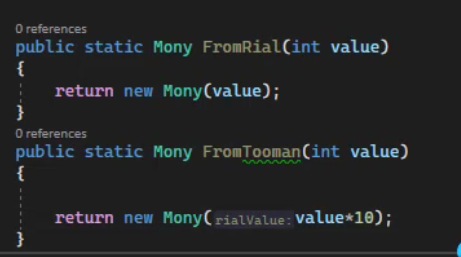
در لایه Domain قرار دارد.

مثال:

در یک entity مقدار price را به جای int از نوع Mony در نظر می گیریم و برای این که immutable باشد private set کرده و فقط در ctor مقدار آن را می گیریم و باید validation ها هم در خودش باشد.

Static Factory method

برای راحت کردن کار نمونه سازی همان کلاس را بر میگرداند و از نوع static است و یک نمونه جدید می سازد.



Aggregate

به مجموعه ای از اشیا به هم مرتبت گفته می شود که جهت کنترل تغییرات به عنوان یک واحد در نظر گرفته می شوند منظور از مجموعه اشیا چندین کلاس است که در یک واحد قرار می گیرند.

سعی کنید aggregate ها را کوچک نگه دارید 1 الی 2 تا

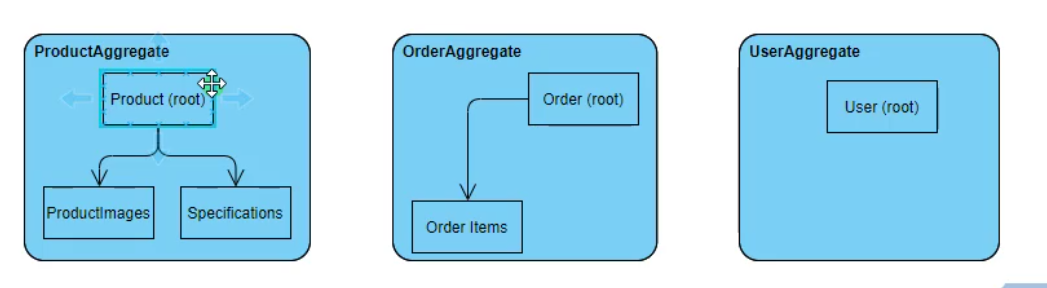
Aggregate root

شی است که قرار است زیر مجموعه های خود را نگه داری کند.

Child ها باید توسط root خود ایجاد شوند.

در این نوع طراحی relation نداریم بلکه روابط بین aggregate ها است و فقط می تواند Id جداول وابسته را بگیرند بدون این که reference داده شوند

نکته : ای دی child را فقط root خود می تواند داشته باشد.



-- Aggregate root مسئول چک کردن invariant ها می باشد. Invariant ها قوانین ساده ای هستند که در entity ما قرار دارند قوانین مربوط به child در root باید وجود داشته باشند و قوانین کسب و کار در root باید باشد..

--اشیا خارج از مرز aggregate نمی توانند به شی غیر از aggregate root دسترسی داشته و یا به ان اشاره کنند یعنی خارج از مرز aggregate چیزی نمی بیند و فقط از تطریق id می تواند دسترسی داشته باشد.فقط یک repository برای یک aggregate داشته باشیم و child ها از همان پیروی کند.

--برای query گرفتن فقط باید از root استفاده کنیم و به چایلد می رسیم و تراکنش های دیتابس در سطح aggregate باید معنی پیدا کند. عملیات insert در یک تراکنش باشد.

اشیا داخل خود aggregate و خود آن می توانند به aggregate root های دیگر اشاره کنند و این اشاره توسط identity انجام می شود.

-- تنها رفرنس قابل قبول در خود aggregate می تواند انجام شود. هرچه روابط کمتر باشد توسعه راحت تر است.

نکته :

زمانی که بخواهیم aggregate ادد کنیم بخواهیم child ها را هم قرار بدهیم در ef کلید اصلی به صورت خودکار اضافه می شود اما در ddd نیاز است id در خود برنامه قرار گیرد معمولا guid می گذاریم و در میکروسرویس نیز پرکاربرد است

راه حل الگوی high low است که یک id سمت Db یکی سمت app ایجاد می کند.