1. Domain driven design چیست؟
   1. تعریف ساده اش اینه: شیوه ای برای توسعه سیستم های پیچیده. منظور اینه که ما یه سری مسئله سمت طراحی و ایجاد سیستم مورد نظرمون داریم که میخوایم اینا رو برطرف کنیم. DDD میاد و یه سری مفاهیم، راهنما و دستورالعمل معرفی می کنه که بر اساس اونا  
      code base مونو ایجاد کنیم.
2. منظور از domain اصلا چیه؟
   1. کلیتش اینه: یه موضوعی وجود داره که حول محور اون میخوایم یه برنامه بنویسیم. Domain میاد و میگه که اطلاعات و دانش مرتبط با اون موضوع و منطق برنامه (application logic) به چه صورته؛ اینکه چه چیز هایی داریم، چه اتفاقاتی میفته، چه اتفاقاتی ممکنه بیفته، در مواجهه با هر اتفاق چه تصمیمی اتخاذ کنیم و...
   2. یه مثال میزنیم: فرض کنیم یه برنامه meet داریم. یه سری افراد دارن داخل این برنامه یه جلسه یا میتینگ برگزار می کنن. هر کس که حضور داره میتونه یه وضعیتی داشته باشه مثل online, disconnect, early left, …. توی هر کدوم از این وضعیت ها ما میتونیم دقیق مشخص کنیم که اگه شخص قرار بگیره، چه اتفاقاتی باید بیفته. حالا مثلا early left رو در نظر بگیریم؛ طرف تا چه مدت زمانی از میتینگ بیرون بره، ما توی این وضعیت نگه اش می داریم؟ یه دیقه، دو دیقه، ده دیقه یا هر مقدار دیگه ای. این ویژگی و تصمیمی که براش می گیریم، همه در حیطه domain هستن.
3. مزایای استفاده از DDD چیه؟
   1. یه متدولوژی ثابت شده اس در طراحی سیستم های پیچیده به خاطر دستور العمل ها و راهنمایی های گام به گام اش در طراحی و پیاده سازی سیستم ها.
   2. در طولانی مدت، سیستم ما انعطاف پذیری بیشتری داره؛ راحتتر میتونیم تغییرات رو بهش اضافه یا اعمال کنیم.
   3. از معماری هایی مثل میکروسرویس یا سیستم های توزیع شده پشتیبانی می کنه (microservice and distributed systems)
   4. تست نویسی به مراتب راحتتره (نداشتن وابستگی های infrastructure و مشخص بودن مرز بین قسمت های مختلف)
   5. چون قسمت های مختلف domain رو جدا و هر بخش رو مجزا هندل می کنه، نگهداری (maintenance) پروژه راحتتره.
   6. گرچه در ابتدا به خاطر پیچیدگی و نداشتن پایه اولیه پروژه، پیاده سازی و توسعه طولانی تر و سختتره اما در طولانی مدت ساده تر و کم هزینه تر میشه.
4. 2 بخش اصلی DDD چی هستن؟
   1. Strategic design: این بخش گرچه به نسبت بخش بعدی کمتر به چشم میاد، اما به شدت مهمه. تو این بخش ما domain رو بررسی می کنیم و سعی میکنیم بفهمیم subDomain های ما چیا هستن، چه ارتباطی با هم دیگه دارن، چرا اصلا وجود دارن (در واقع میخوایم بدونیم هر subDomain چه نقش کلیدی ای رو داره بازی میکنه که بودنش رو واجب میکنه)، هر subDomain رو کجا قرار میدیم (ممکنه چندتاشون با هم داخل یه برنامه باشن و یک یا چندتا داخل یه برنامه دیگه)، میزان اهمیت و هزینه توسعه هر subDomain و... . به طور کلی، میپرسیم که چه چیزهایی داخل domain ما هستن و چرایی وجودشون؛ اینکه چه مشکلی رو دارن برای ما حل می کنن که به خاطرش بخوایم اینا رو داشته باشیم. در نهایت، هر subDomain باید به یه logical self-contained component تبدیل بشه که در درون خودش، داره مسائل مربوط به اون subDomain رو حل می کنه و با subDomain های دیگه هم در ارتباطه.
   2. Tactical design: اینجا حالا با "چطور" سر و کار داریم. چطور یه subDomain کار میکنه؟ چطور با subDomain ها ارتباط برقرار میکنه؟ چطور میخواد مسائل رو حل کنه؟ جواب این سوال ها رو توی این بخش میدیم که بیشتر با مفاهیم کدنویسی و فریمورک هاش سروکار داره.
5. منطور از ubiquitous language در DDD چیه؟
   1. یه زبان مشترک درباره domain که توسط همه افراد تیم توسعه دهنده استفاده میشه تا بتونن به صورت واضح و پایدار (consistent) بر طبق اون با هم دیگه در ارتباط باشن. منظور اینه که یه سری تعاریف و کلیات هست که وقتی کسی درباره اونا صحبت میکنه، نیازی به رمزگشایی و توضیح نیست و همه متوجه میشن منظور چیه.
   2. این زبان بسته به domain یا subDomain میتونه فرق کنه؛ ممکنه یه مفهوم داخل یه subDomain به یه شکل تعریف بشه و معنی بده و در یه subDomain دیگه تعریف و معنی دیگه ای داشته باشه. اما در هر حال، وقتی مشخص باشه که اون مفهوم کجا مطرح داره میشه، طرف مقابل متوجه میشه که تعریف و معنی مورد نظر، کدوم هست.
   3. این زبان فقط به کدنویس ها محدود نمیشه و افرادی مثل تحلیلگر هم با این زبان مشترک دارن درباره domain صحبت می کنن.
6. Invariant ها چی هستن؟
   1. اینا Business rule ها هستن که همیشه باید برقرار باشن. مثلا توی همون اپ میتینگ، قرار میذاریم که تعداد افراد شرکت کننده داخل یه میت نباید بیشتر از 30 تا باشه (حالا به هر دلیلی) یا هر میت نباید بیشتر از 60 دقیقه طول بکشه. اینا یه سری قواعد هستن که قرار نیست شکسته بشن و همیشه برقرار هستن (یکم شبیه به validation هایی هستن که به طور عادی روی انتیتی ها میذاریم).
7. منظور از Enforcing invariants چیه؟
   1. ما داخل domain مون یه سری انتیتی (domain model) داریم. Invariant ها هم که business rule ها هستن. Enforcing invariant یعنی ما بیایم این business rule ها رو روی انتیتی هامون اجبار کنیم؛ در واقع همیشه انتیتی ها تحت هر شرایطی از اون invariant ها تبعیت کنن. حالا این کار رو به دو صورت میشه انجام داد: یا یه جایی خارج از خود انتیتی این اجبار انجام بشه یا اینکه داخل خود انتیتی این اجبار اتفاق بیفته.   
      در حالت اول این invariant ها (که domain logic هم بهشون میگیم) مستقیما داخل انتیتی نمیشینن بلکه جایی خارج از کلاس اون انتیتی داره این domain logic پیاده سازی و enforce میشه.  
      در حالت دوم (که DDD میگه این رو باید لحاظ کنیم) ما داخل کلاس انتیتی این  
      domain logic ها رو پیاده سازی می کنیم. در واقع domain logic میره داخل domain model و نه خارج از اون.
8. Result pattern چیه؟
   1. ما یه زمانی هست که میخوایم اعلام کنیم که چیزی موفقیت آمیز بوده یا نبوده. اگه بوده که صرفا اعلام اوکی بودن می کنیم اما اگه نبوده، باید اینم بگیم که چرا نبوده.  
      با این pattern، ما میتونیم هر دو نوع موفقیت آمیز بودن و نبودن رو به عنوان یه نوع تعریف کنیم در حالی که خود موفقیت (Success) و عدم موفقیت (Error) هر کدوم شئ جدا باشن.  
      اینطوری انگار همیشه داریم برای مثلا یه متد، یه مدل خروجی میدیم.
   2. این pattern برای خروجی API ها و تست نویسی خیلی مناسبه.
9. Tactical pattern؟
   1. قبلا درباره tactical design توضیح دادیم، اینم ادامه همون میشه. به این صورت که پایه و اساس نحوه انجام و پیاده سازی پروژه رو با tactical pattern انجام میدیم. بخش های مختلف tactical pattern که پایه های نحوه پیاده سازی ان (درباره هر کدوم مفصل صحبت می کنیم) میشن:
      1. Entities
      2. Value objects
      3. Aggregates
      4. Domain services
      5. Factories
      6. Repositories
      7. Domain events
10. Entity چیست؟
    1. کلاسی که ویژگی های زیر رو داشته باشه میتونه یه entity باشه:
       1. Identifier: چیزی که باهاش بشه هر نمونه از اون کلاس رو شناسایی کرد (معمولا فیلد ID توی کلاس اگه باشه، identifier ما اون به حساب میاد)
       2. Equality: دو شئ از اون کلاس بر چه اساسی با هم برابر میشن؟ چه فیلد هایی هستن که با مقایسه اونا دو شئ برابر میشن یا نه و یا چه منطقی به کار میره برای سنجش برابری دو شئ از یک کلاس؟ این میشه بحث equality که کلاسی که میخواد entity باشه، باید بتونه برابری دو شئ از خودش رو تعیین کنه.  
          به طور عام، برابری با فیلد ID سنجیده میشه و حتی اگه دو شئ از اون کلاس باقی فیلد هاشون فرق کنه، صرف اینکه ID های برابر داشته باشن از نطر ما برابر به حساب میان. از این رو معمولا base class ای که برای entity میسازیم، داخلش متد Equals به طور پیش فرض override شده و داره ID دو شئ از کلاس رو با هم مقایسه میکنه. اگه لازم باشه، توی خود کلاس میتونیم این متد رو مجددا override کنیم.  
          مشابه این کار رو برای متد GetHashCode هم انجام میدیم.
11. Value object چیست؟
    1. اینجا ما یک یا مجموعه ای از پراپرتی ها رو داریم که در کنار هم یه معنی بزرگ تری رو میدن. مثلا کلاس Email با یه پراپرتی Address که صرفا داره مشخص میکنه این پراپرتی منظورش آدرس ایمیله؛ حالا چرا به جای اینکه یه پراپرتی string به اسم Email داشته باشیم، خود Email رو یه کلاس جدا تعریف میکنیم؟ دلیل اینه که ما یه سری کار ها داریم که مخصوص مفهوم Email هست (مثلا اینکه باید از یه فرمت خاصی پیروی کنه). به جای اینکه بیایم این کار ها رو ببریم داخل کلاسی که میخواد یه پراپرتی برای Email داشته باشه، کلا مفهوم Email رو تبدیل به کلاس می کنیم و رفتار های مختص خودشو داخل خودش تعریف می کنیم.  
       یا مثلا مفهوم TimeRange که میتونه از دو پراپرتی از نوع DateTime به اسم های StartDate و EndDate تشکیل بشه. حالا رفتار هایی مثل همیشه بزرگتر بودن EndDate از StartDate یا بدست آوردن فاصله بین شروع و انتهای یه TimeRange، همه و همه میتونن داخل یه کلاس باشن که نوع TimeRange رو با این دو پراپرتی و تمام رفتار های مربوط به خودش، هندل میکنه
    2. احتمالا با این اوصاف سوال پیش میاد که تفاوتش با Entity چیه که جواب اینه که ValueObject دیگه Identifier نداره. از طرفی، یه ValueObject مستقلا استفاده نمیشه و خودش جزئی از یه Entity هست.
    3. یه ویژگی دیگه شون اینه که Immutable هستن؛ یعنی مثلا ساختن یه شئ از یه ValueObject با مثلا new کردن اش اتفاق نمی افته بلکه هر گونه تغییر یا تاثیری که قراره روی اون ValueObject گذاشته بشه، از طریق متد هایی هست که داخلش می نویسیم و اجازه دسترسی از بیرون میدیم. مثلا آپدیت کردن یه شئ از یه ValueObject دیگه با تغییر دادن پراپرتی هاش امکان پذیر نیست و باید متد Update ای براش وجود داشته باشه و اونو کال کنیم.
12. Aggregate چیست؟
    1. یه تعریف انگلیسی هس که هرچی زور میزنم فارسیش در نمیاد خودشو میذارم:  
       One or more domain objects that always need to stay consistent as a whole  
       منظور چیه؟ ما دو مدل domain object داریم: Entity , Value object. اما یه سری وقتا هست که یک یا چندتا از اینا رو یه جا لازم داریم اما این یه جا، خودش مستقیما یه Entity یا value object نیست.
    2. با یه مثال توضیح میدیم: فرض کنیم داریم یه میتینگ ست می کنیم. اینجا entity ای به اسم Reservation رو داریم. حالا در ظاهر باید یه انتیتی دیگه به اسم Session بسازیم که داخل خودش داره لیستی از Reservation رو نگه میداره اما در اصل Session بیشتر از اینا داره کار انجام میده؛ داخلش علاوه بر List<Reservation>، برگزار کننده و حداکثر تعداد شرکت کنندگان رو هم نگه میداره. پس در اصل خودش از انتیتی های دیگه تشکیل میشه.
    3. از طرفی رفتاری که Session بروز میده هم مهمه. اگه یه Session داریم، همیشه در یه وضعیت valid هست. مهم نیست که عملیات اضافه کردن یه نفر به میتینگ موفقیت آمیزه یا نه، خود میتینگ قبل و بعد از این عملیات بوده و همچنان پابرجاس (وضعیت valid) و این داره اون بخش consistent توی تعریف رو توضیح میده.
    4. برای اینکه مشخص باشه که چه انتیتی هایی از نوع Aggragate هستن، یه base class برای Aggregate ها می سازیم که هر Aggregate ای ازش ارث بری کنه. این کلاس معمولا خودش داره از Entity ارث بری می کنه اما برای ما مشخص می کنه که کلاسی که داره ازش ارث بری می کنه، یه Aggregate هست.
13. Domain service ها چی هستن؟
    1. یه سری سرویس هایی هستن که برای domain مون می نویسیم. اینا قراره که  
        business logic و business rule ها رو پوشش بدن منتهی سوال پیش میاد که مگه ما اینا رو داخل domain object هامون نذاشتیم؟ پس چرا باید به همچین چیزی نیاز داشته باشیم؟  
       جواب اینه که در اصل فایده domain service ها اینه که گرچه برای پوشش دادن business rule & logic به کار میرن اما در واقع اون هایی رو پوشش میدن که مستقیما نمیشه داخل domain object ها گذاشت. در اصل وقتی که داریم قواعدمون رو پیاده سازی می کنیم، احتمالش هست که یه سری هاشون رو نشه مستقیما داخل domain object ها نوشت؛ به جاش میایم داخل domain service ها می نویسیم
    2. به ندرت پیش میاد که به domain service ها نیاز پیدا کنیم. معمولا اینطوریه که اگه حس کردیم بهشون نیاز داریم، یه مقدار لازمه فکر کنیم که آیا واقعا نیاز هست یه domain service تعریف کنیم یا با کمی تغییر توی domain object ها میشه اونجا هندلش کرد؟  
       اما به این معنی نیس که اصلا بهشون نیاز پیدا نمیکنیم.
14. Factory ها چی هستن؟
    1. به طور ساده، مسئول ایجاد اشیا پیچیده هستن. وقتی یه کلاس رو داریم که ایجاد کردن شئ ازش، پیچیده میشه و کار های زیادی باید قبل از ساختش انجام بشه یا validation هایی پاس بشه، میتونیم بیایم یه کلاس یا متد Factory تعریف کنیم و ساخت شئ از اون کلاس رو ببریم داخل این متد.
    2. اینکار علاوه بر اینکه منطق ساخت شئ رو ایزوله میکنه (و دیگه هیچجوره نمیشه غیر از اون متد، شئ از کلاس مورد نظر ساخت)، به سادگی و خوانایی کد هم اضافه میکنه؛ ما دیگه لازم نیست داخل خود کلاس اصلی به خاطر کد زیاد، سردرگم بشیم. در ضمن به ما امکان استفاده از Result pattern رو هم میده که به این درد میخوره که اگه در حین ساخت شئ از کلاس مورد نظر، به هر دلیلی خطایی پیش اومد، کلا شئ ای ساخته نشه و یه Error پاس داده بشه به جای Exception raising. اگر هم شئ ساخته شد، خودش پاس داده میشه.
    3. معمولا یه static method داخل همون کلاس هستن که بهتره constructor کلاس رو private تعریف کنیم تا کلا کسی نتونه به غیر از این متد، جور دیگه ای شئ از کلاس بسازه.
15. Repository ها چی هستن؟
    1. یه پکیج از سرویس هایی هست که از بیرون میشه روی domain object ها (توی این مورد، بیشتر انتیتی ها و Aggragate ها) اعمالشون کرد. یکم گیج کننده اس نه؟  
       ما اگه لازم داشته باشیم مثلا یه رکورد از کلاس رو بر اساس Id اش پیدا کنیم و بیاریم، میتونیم کدی که قراره اینکارو انجام بده ببریم داخل repository و بذاریم داخل یه متد مثل GetById. یا اگه نیاز باشه که یه رکورد ما آپدیت بشه، میتونیم کدشو مثل GetById ببریم داخل repository و یه متدش کنیم.
    2. این پکیج که به بیرون از domain object اجازه دسترسی و تغییر و... رو روش میده، میگیم repository.  
       به طور معمول repository ها عملیات های CRUD رو روی domain object ها هندل میکنن.
    3. مشابه factory، اینکار تغییر دادن روی domain object ما رو به متد ها و شیوه ای که ما صلاح میدونیم، محدود میکنه و دیگه کسی نمیتونه خارج از این چارچوب روی domain object عمل کنه.
16. Disconnected domain model ها چی هستن؟
    1. وقتی که داخل یه aggregate نیازه که با یه انتیتی یا aggregate دیگه ای کار کنیم و از repository اون انتیتی یا aggregate برای ارتباطش با aggregate مورد نظر استفاده می کنیم، بهش گفته میشه disconnected domain model. چرا که دیگه خود اون انتیتی یا aggregate داخل domain استفاده نشده. چنین اتفاقی یه زنگ خطره به این معنی که احتمالا طراحی ما مشکل و نیاز به بازبینی داره.