جزوه درس "تحلیل و طراحی سیستم ها" / استاد مربوطه: دکتر مهدی مجد

جلسه 1

قبل از شروع درس ، در مورد معنی اسم این درس صحبت میکنیم.

معادل انگلیسی " تحلیل و طراحی سیستم " ، System Analysis and Design است. که Analysis معادل "تحلیل" و Design معادل "طراحی" است.

در مکالمات رسمی به جای واژه "سیستم" از معادل فارسی "سامانه" استفاده میکنیم.

تعریف سیستم: مجموعه ای از اجزا و رابطه ها است. (یک سری اجزا که باهم رابطه و فعالیت و همکاری دارند)

که بین برخی از افراد یک اختلافی وجود دارد ، اینکه یک سری میگویند سیستم ، مجموعه ای از اجزا که با همکاری هم برای رسیدن به یک هدف فعالیت میکنند. که عده ای دیگر هدف را از سیستم جدا میکنند و علت این جدایی هدف از سیستم را اینگونه بیان میکنند که هر سیستمی ممکن است با توجه به ناظری که داره به اون سیستم نگاه میکنه یا از آن سیستم استفاده میکنه ، ممکن است هدف متفاوتی داشته باشد. (***هدف یک سیستم وابسته به ناظر است.***)

پس برای یک سیستم نمیتوان یک هدف واحد مشترک در نظر گرفت چون بسته به نگاه و استفاده ی ناظر ، میتواند هدف متفاوتی مشخص شود. (*پس هدف رو از سیستم جدا میکنیم.*) !!!

به عنوان یک مثال کوچک ، صندلی یک سیستم است. چون در آن مجموعه ای از اجزا با هم در ارتباط اند ولی هدف استفاده از آن بستگی به ناظر آن دارد. چون ممکن است یک نفر از آن به

عنوان نشیمنگاه یا ممکن است یک نفر از آن به عنوان پایه ای برای نگه داشتن در اتاق و کلاس استفاده کند و... . *پس تفاوت آن در هدف است که بسته به ناظرآن میتواند متفاوت باشد.*

پس نتیجه ی نهایی :

سیستم مجموعه ای از اجزا و رابطه ها است که هدف و کارکرد آن با توجه به ناظر آن سیستم ، میتواند متفاوت باشد.

سیستم = مجموعه ای از اجزا + رابطه ها

relationship (R) element set

این اجزا میتوانند هرچیزی (thing) باشند. (مثل X , y)

هرچیزی (thing) میتواند object باشد.

اجزا جامع ترین اسمی است که برای هرچیز میتوان استفاده کرد.

S = (E , R)

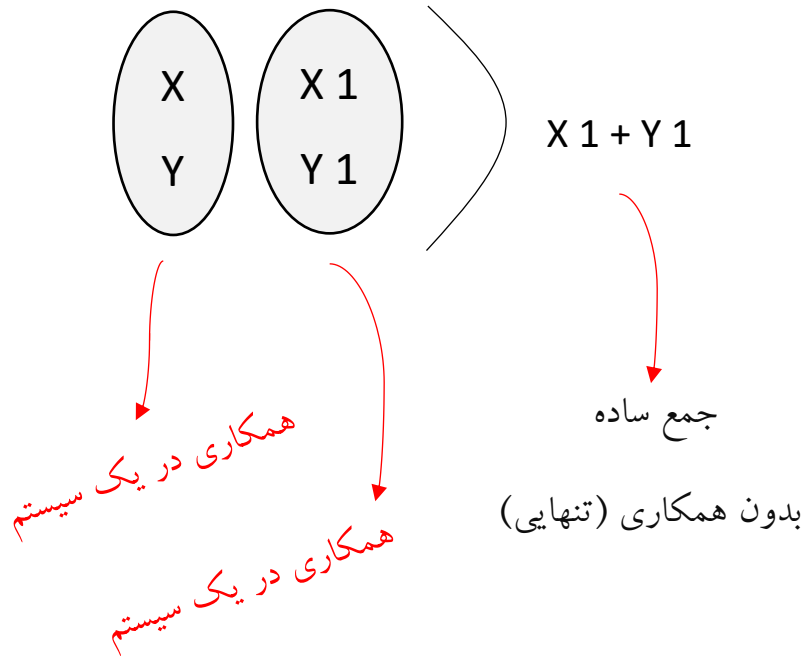
این R ، همکاری 2 جز X , y رو نشان میدهد. که این رابطه به اون سیستم بستگی دارد.

یکی از مزایای ایجاد همکاری بین اجزا ، هم افزایی است.

توضیح هم افزایی:

فرض کنید 2 اجزای X , Y را داریم. که ارزش ، سود یا فایده ی جزء اول را X 1 و سود جزء دوم را Y 1 میگوییم. در کنار هم قرار گرفتن اجزا ، ارزش بیشتری تولید میکند نسبت به تنهایی هر جزء. (همکاری اجزا با هم ، ارزش افزوده ی بیشتری تولید میکند از ارزش هرکدام به تنهایی.)

هم افزایی:



Sub system / زیر سیستم

سیستم های بزرگتر به بخش های کوچکتری تقسیم میشوند که به آن ها زیر سیستم اون سیستم های بزرگتر میگویند.

هر زیر سیستم میتواند برای خودش یک سیستم مستقل باشد ولی در هر صورت در دل اون سیستم بزرگتر است.

مطالبی که در بالا اشاره شد ، جز درس اصلی نیستند !!!

خب ، وارد مبحث این درس میشویم.

هدف ما در این درس این است که میخواهیم یک سیستم نرم افزاری تولید کنیم که دستورات کارفرما را انجام داده و گزارش های مورد نظر را ارائه دهد. (میخواهیم یک سیستم فعلی را تحلیل و بررسی کنیم)

یک سیستم فعلی میتواند موجود باشد. (مثلا یک آموزشگاه است)

گاهی اوقات یک سیستم موجودی نداریم ، و آن سیستم **به صورت تحلیلی** در ذهن فرد هست.

تحلیلی به این معنی نیست که آن سیستم وجود ندارد. بلکه به این معنی است که آن سیستم هنوز به مرحله ی پیاده سازی و عملیاتی نرسیده است.

سیستم فعلی

- **موجود** (نرم افزاری است.)
- **تحلیلی** (یعنی سیستمی که هنوز پیاده سازی و عملیاتی نشده است و فعلا فقط در ذهن است.)

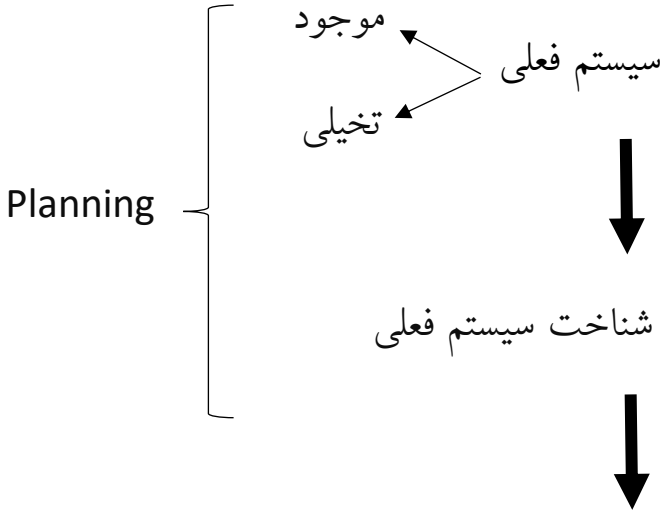
در تحلیل یک سیستم فعلی **اولین کار ، شناخت سیستم است.** (پیدا کردن شناخت درستی از سیستم و مطالعه در مورد آن و یا کمک از تجربه های پروژه های قبلی)

و **در مراحل بعد** ، به نیازهای اون سیستم و **شناخت نیازهاش** میپردازیم.

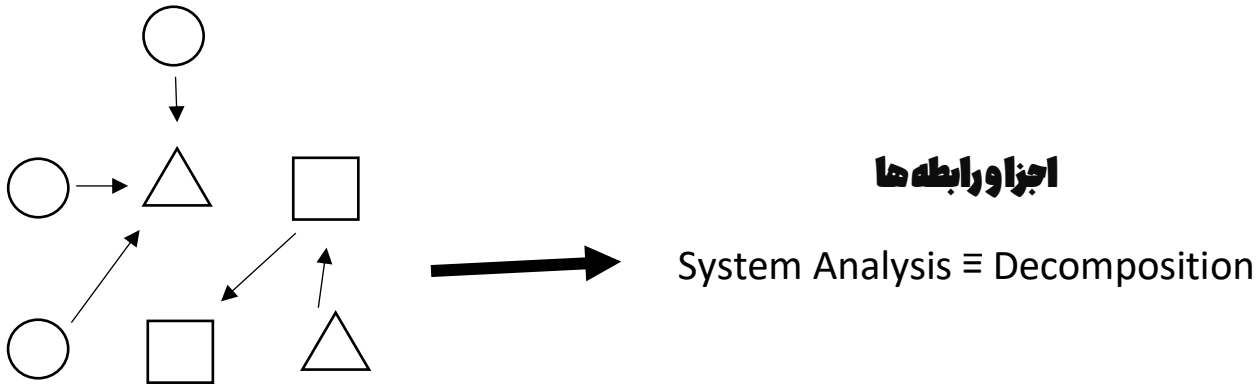
بعد از شناخت سیستم ، ما میتوانیم اجزا و رابطه ها را بشناسیم.

تحلیل سیستم، یعنی بعد از اینکه یک سیستم را شناختیم ، آن را به اجزا و رابطه هایی تقسیم کنیم.

(یعنی شکستن یک سیستم به اجزا و رابطه ها)



System Analysis (تحلیل سیستم) : شکستن و تفکیک یک سیستم به اجزا و رابطه ها (Decomposition).



بعد از Decomposition ، برای **پیاده‌سازی** یک سیستم جدید بر اساس سیستم فعلی ، ممکن است ما نخواهیم تمام اجزای سیستم قبلی را در سیستم جدید بیاوریم. چون ممکن است سیستم جدید ، تنها بخشی از سیستم فعلی را داشته باشد یا درواقع مدلی از آن را برداشت میکنیم. (نه همه)

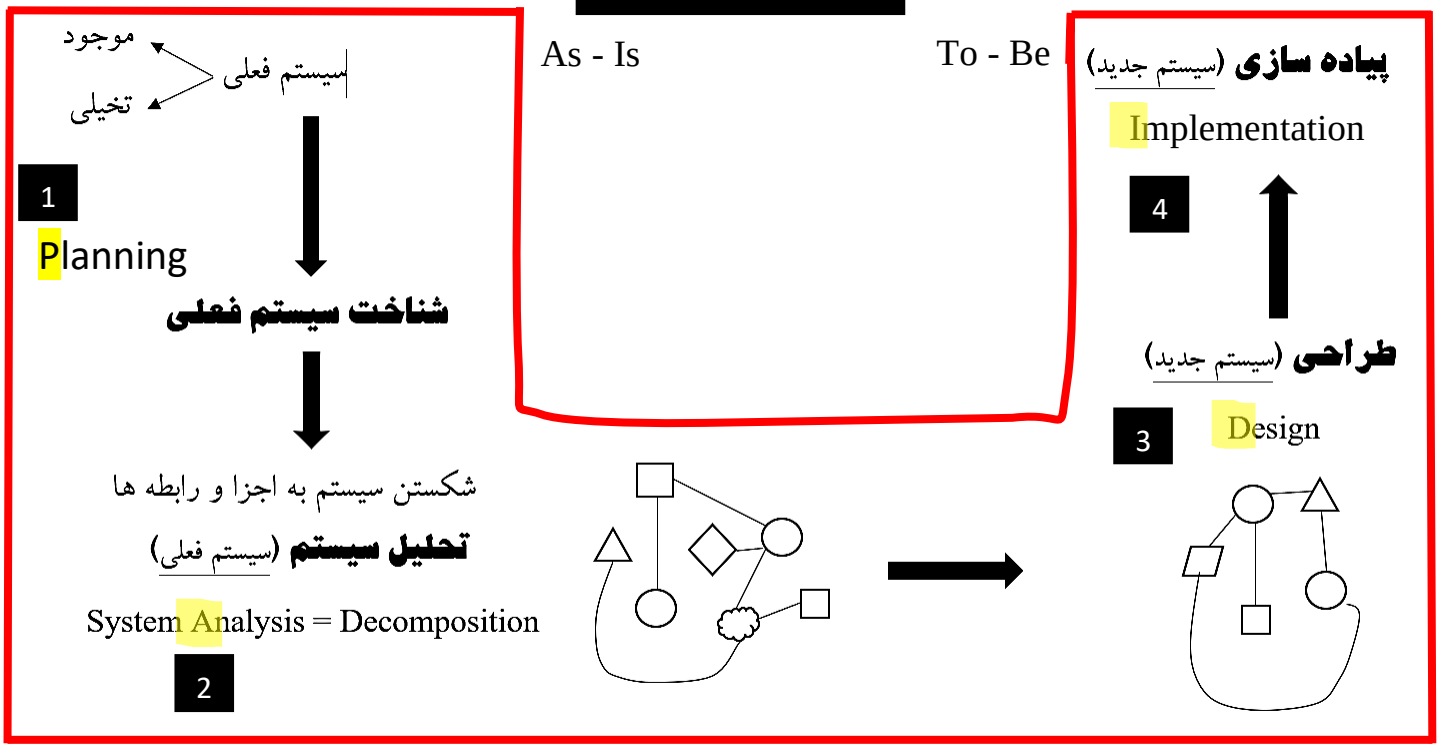
پس در طراحی سیستم جدید ، ممکن است همه ی اجزای سیستم اولی (فعلی) را استفاده نکنیم.

و حتی برای طراحی سیستم جدید ، ممکن است روابط بین اجزاها را تغییر دهیم و حتی اجزا و روابط جدیدی هم ایجاد میکنیم. یا فقط روابط قبلی را اصلاح میکنیم.

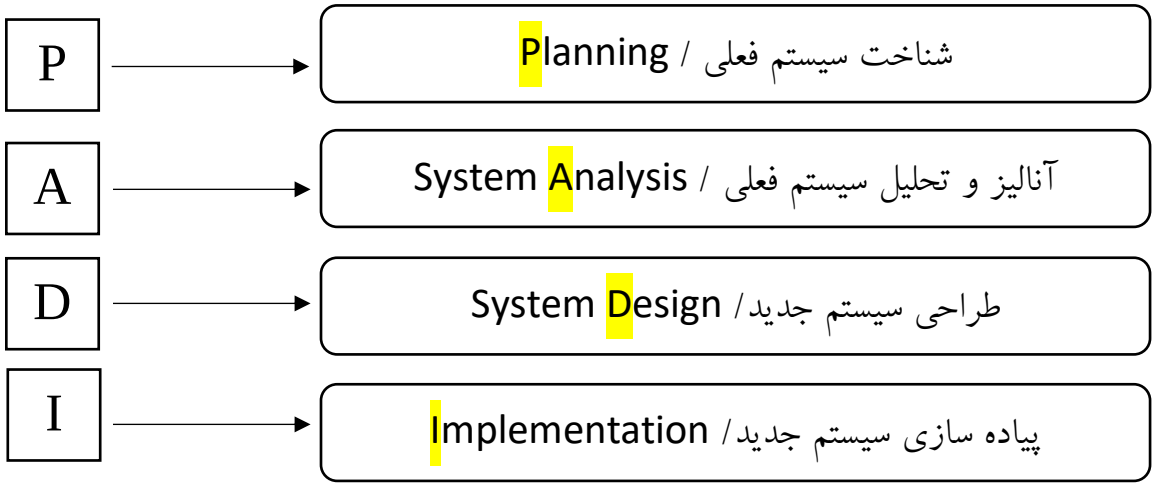
که این تغییرات برای طراحی یک سیستم جدید باید عالمانه و با تفکر و دقیق باشد.

گاهی اوقات هم ، سیستم جدید همان سیستم فعلی است که دچار تغییرات و اصلاحاتی شده است. یعنی مثلا ما همان سیستم فعلی را میخواهیم با این تفاوت که قصد داریم یک سری از اجزا حذف یا روابط بین آن ها دچار تغییراتی شود. (مثل اینکه یک اپلیکیشن وجود دارد (سیستم فعلی) و سازندگان اون اپلیکیشن ، قصد به روز رسانی آن را دارند (سیستم جدید). پس سیستم جدید هم همان سیستم فعلی بوده که تنها یک سری تغییرات در اجزا و روابط داشته است.)

شکل زیر که به آن "مدل U شکل" میگویند ، یک نمای کلی از مرحله ی شناخت سیستم فعلی تا پیاده سازی سیستم جدید است. که شامل 4 مرحله است.



4 مرحله ی ساخت سیستم جدید از سیستم فعلی:



مرحله "**پیاده سازی**" ، شامل مراحل متفاوتی از جمله کدنویسی ، آزمایش های خطایابی ، پشتیبانی و ... است که آنها در درس مهندسی نرم افزار بررسی میشوند و هدف این درس ما نیستند / و هدف ما در این درس ، درمورد تحلیل و طراحی یک سیستم نرم افزاری است که در مورد جزییات آن صحبت میکنیم. (رسیدن از سیستم فعلی به سیستم جدید)

برای سیستم فعلی ، میتوان اسامی مختلفی مانند As - Is **یا** Business **یا** سیستم موجود **یا** سیستم قبلی **را در نظرگرفت.**

As - Is به معنای سیستمی است که الان وجود

Business یعنی آن فعالیت یا تجارتی که الان در حال انجام است.

به سیستم جدید To - Be **میگویند**.

To - Be یعنی آن سیستمی که در آینده به وجود می آید و باید به آن برسیم.

جلسه 2

داستان و سرگذشت مهندسی شدن تحلیل و طراحی سیستم ها

مطلب خارج از مبحث درس (بیشتر بدانیم)

در گذشته ، یعنی **قبل از** تولید روش های مهندسی و مدون ، تولید سیستم های نرم افزاری یک حالت متکی به شخص را در تولید داشتند. یعنی تولید سیستم های نرم افزاری یک هنر تلقی میشد.

چون افراد با فکر به چگونگی طراحی یک سیستم و با سلیقه های شخصی خود ، یک سیستم را تولید میکردند. بنابراین کارهای یک فرد یا شرکت ، با کارهای یک فرد یا شرکت دیگر کاملا متفاوت بود.

کارهای تحلیل و طراحی یک سیستم متکی به شخص بود ، **بنابراین خطاها به سختی کشف می شد.**

و نتیجه ی آن این شد که این بحران ها روی هم جمع شدند و در حدود دهه 80 تا 90 میلادی ، یک بحران بزرگ را به وجود آوردند.

آن ها سیستم ها را طراحی میکردند و نرم افزاری را به وجود می آوردند ولی این نرم افزار بخاطر درست طراحی نشدن آن ، بعد از پیاده سازی و مورد استفاده قرار گرفتنش ، تازه خطاهای آن پیدا میشد. و شروع به برطرف کردن خطاهای سیستم های فعلی کردند که همین کار باعث شد از درخواست های جدید عقب بمانند و در نتیجه سبب شد کارها روی هم جمع شوند.

که درنهایت به این فکر رسیدند که تحلیل و طراحی سیستم ها را از یک حالت هنری ، شخصی و سلیقه ای به یک کار مهندسی و مدون تبدیل کنند ، که در آن دیگر روند انجام کار وابسته به شخص و سلیقه ای نباشد.

شروع مبحث جلسه 2

رسیدن از سیستم فعلی به سیستم جدید ، از تعامل بین طراح و کارفرما به دست می آید.

که باید طرفین با هم گفت و گو داشته باشند تا طراح بتواند شناخت درستی از سیستم فعلی پیدا کند که در نهایت باعث کاهش یا عدم خطا شود تا بتواند رضایت طراح را جذب کند.

در طراحی یک سیستم جدید ، باید از آن 4 مرحله ای که در صفحات قبل توضیح داده شده استفاده کنیم. **اما** نوع ترتیب قرارگیری و شیوه ی استفاده از آن 4 مرحله ، با توجه به هر سیستم متفاوت است و درواقع بستگی به آن سیستم و جزییات آن دارد.

پس باید برای انجام این مراحل ، ترتیبی را در نظر بگیریم که شناخت این ترتیب به آن کاری که میخواهیم انجام بدیم وابسته است.

مثال: فکر کنید که یک مسیری وجود دارد که ما تا به حال از آن مسیر عبور نکرده ایم ، و به گفته شده که یک آب جاری در این مسیر وجود دارد و ما باید از این مسیر عبور کنیم.

در ابتدا باید آن را بشناسیم. یعنی اینکه این آبی که در مسیر ما وجود دارد ، در چه حد و اندازه ای است یا ارتفاع آن چقدر است و... . که ما با توجه به شرایط آن آب ، به فکر شنا یا جلیقه یا ساختن پل با چوب و... باشیم یا حتی به راحتی میتوان از آن عبور کرد یا از آن پرید. (شناخت درست آن با جزییات)

در اینجا هم دقیقا به همین صورت است. یعنی اون سیستمی که میخواهیم طراحی کنیم ، تعیین میکنه که ما چطور این مسیر را طی کنیم.

در تمام این کارهای مختلف ، ما نمیتوانیم یک روش واحد رو پیش ببریم. برای همین مجبوریم **مدل های مختلف** به کارگیری مراحل را به کار ببریم. که اما در دل همه ی این مدل ها نهایتا این ترتیب به وجود میاد که اول سیستم رو بشناسیم / بعد تحلیل / بعد طراحی / و در نهایت پیاده سازی کنیم.

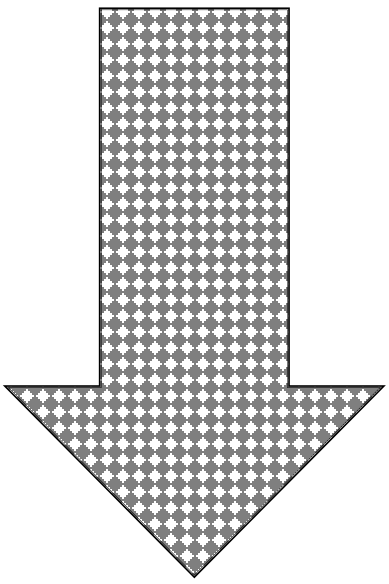
Process Models / مدل های فرآیند

Process Models (مدل های فرآیند): نحوه و ترتیب انجام فعالیت و کارهای مورد نیاز

جهت رسیدن از سیستم فعلی به سیستم جدید است. (اهمیت آن = مشخص کردن ترتیب انجام کارها است.)

درواقع به فرآیندهای دارای ماهیت یکسان گفته میشود که قابلیت طبقه بندی در یک مدل واحد را دارند. مدل های فرآیند دارای کاربردهای فراوانی در مهندسی سیستم ها و... میباشند.

مدل های مختلف فرآیند به شرح زیر اند:



مدل آبشاری (Waterfall)

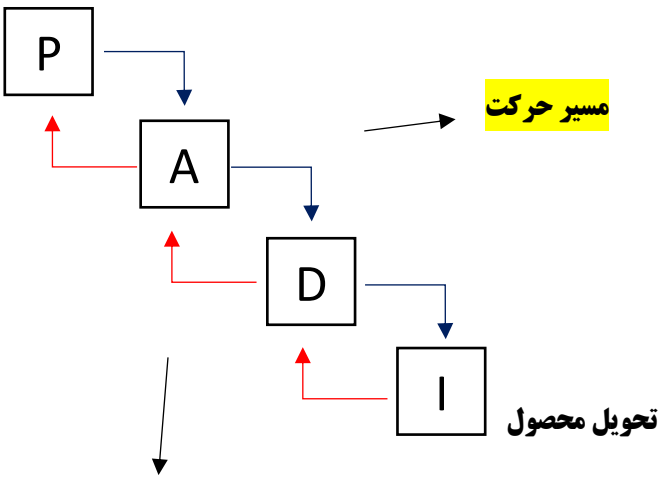
ساده ترین مدل ، مدل آبشاری است.

برای سیستم های ساده و کم حجم ، مدل آبشاری مناسب است.

برای سیستم های کوچک مناسب است ، نه خیلی بزرگ.

بدترین خطا ، خطایی است که در مرحله ی شناخت سیستم بوده و آن را در مرحله ی پیاده سازی متوجه شده ایم.

(عدم شناخت کامل سیستم)



مسیر بازگشتی (برای زمانی که خطایی داشتیم. و اگر خطایی نداشتیم ، خب مسیر بازگشتی هم نداریم.)

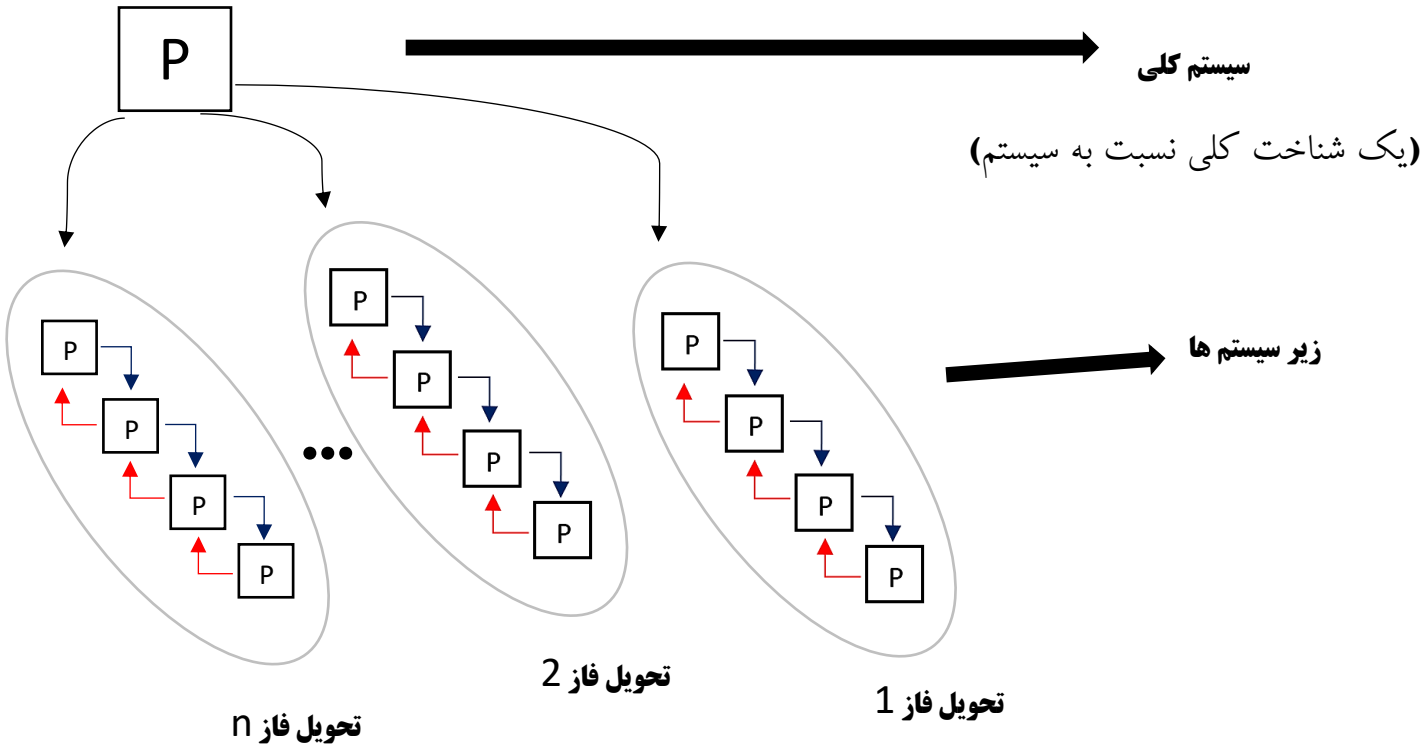
در مدل آبشاری **ابتدا** شناخت سیستم فعلی / سپس تحلیل سیستم فعلی / بعد از آن ، طراحی سیستم جدید / و در نهایت پیاده سازی است. که بعد از اتمام هر 4 تای این مراحل ، محصول نهایی آماده ی تحویل است. (اولین خروجی نرم افزار ، بعد از اتمام 4 مرحله ی بالا است.)

اگر در انجام هرکدام از این مراحل ، متوجه شدیم که خطا و اشتباهی رخ داده ، باید برگردیم (با مسیر بازگشتی) و مرحله ی قبلی را دوباره انجام دهیم.

برای پیاده سازی سیستم های ساده و کم حجم یا برای افرادی که حرفه ای شدن یا کار برای آن ها تکراری شده ، مدل آبشاری مناسب است. که طراح با کارفرما درمورد شناخت آن سیستم صحبت میکنند. در مدل آبشاری تاکید بر دنبال کردن فعالیت ها به ترتیب است.

ولی اگر سیستم ما پیچیده تر بود ، از مدل های دیگری میتوانیم استفاده کنیم.

مدل فازبندی شده (Phased)



در این مدل ، اول یک شناخت کلی نسبت به سیستم موجود پیدا میکنیم و بعد به صورت زیر سیستم ها ، شناخت جزییات با Implementation, Design, Analysis , Planning انجام میشود.

در این مدل هر زیر سیستم را یک فاز میگوییم که هر فاز را جداجدا طی میکنیم.

ممکن است زیرسیستم ها با هم ارتباط داشته باشند.

در این مدل ابتدا یک بخشی از سیستم را طراحی و تحویل میدهیم. در این مدل یگ گروه فاز اول را انجام میده و همان گروه فاز دوم و پس فاز سوم را انجام میدهد.(در مدل فازبندی شده ، یک گروه تمام کار ها را انجام میدهد.)

مثلا: اگر یک کاربر نخواهد کار را به طور کامل تحویل گیرد ، یا مثلا به خاطر کمبود بودجه نیاز داشته باشد برنامه مورد نیازش را جداجدا تهیه کند. در این صورت از مدل فاز بندی شده استفاده میکنیم. در این مدل ، ابتدای هرفاز یکسان ولی تحویل و مسیر آن ها جدا از هم است.

مدل چرخشی (Iterative)

	I 1	I 2	I 3	
P	% 20	% 10		کار 1
A	% 30	...		کار 2
D	% 10	...		کار 3
I	% 5	...		کار 4

مدل چرخشی برای زمانی استفاده میشود که مثلا 20٪ از P را انجام میدهیم و دوباره چند درصد از A و به ترتیب I , D ، و سپس **آن را نشان کارفرما میدهیم** تا ذهنیتی و شناختی از کار برایش ایجاد شود و خطاها کمتر شود. در این صورت برطرف کردن خطاها راحت تر و کم هزینه تر است.

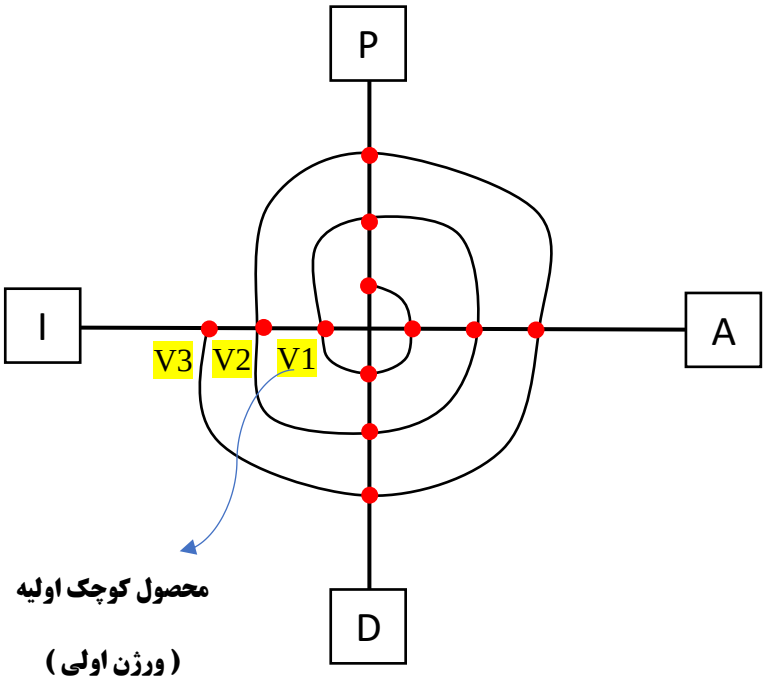
همه ی بخش های I , D , A , P باید 100 درصد کامل شوند ، به ویژه آخرین بخش آن.

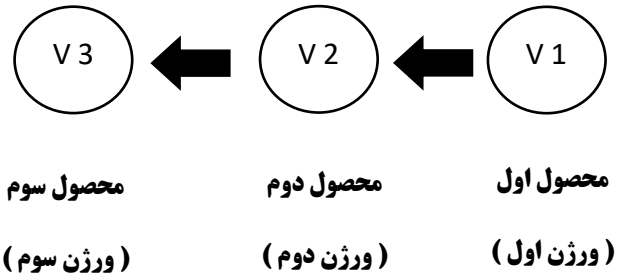
و تا زمانی که بخش آخر کامل شود ، کار آماده نیست.

این مدل برای هماهنگ کردن ذهن و ایده ی کارفرما با طراح است که طراح باید **یک نمونه اولیه** از پروژه به کارفرما تحویل دهد تا کارفرما یک ذهنیت و کلیتی از کار را ببیند.

مزیت این مدل این است که اگر خطایی در مرحله ای رخ دهد ، راحت تر میتوان آن را رفع کرد.

مدل مارپیچی (Spiral)





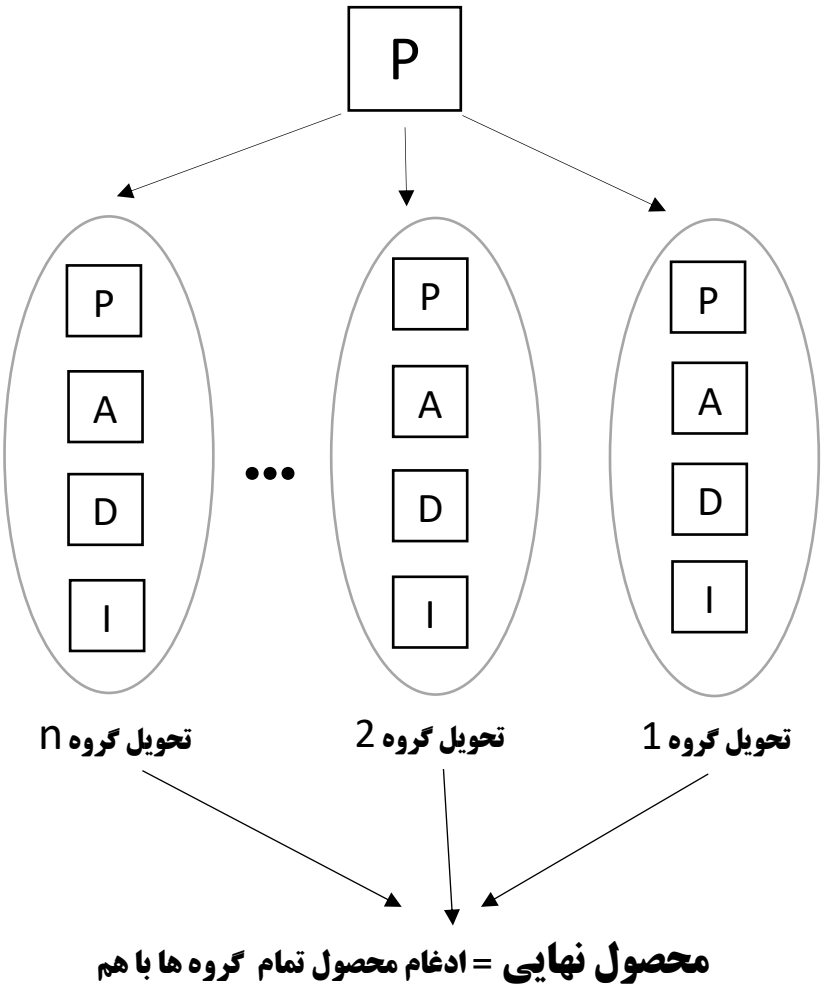
این نوع چینش و مدل ، برای برنامه ای که جز به جز افزایش پیدا میکند است. (مشابه رشد انسان)

درواقع در این مدل میتوان به برنامه یا سیستم ، چیزی اضافه کرد. مثل اپلیکیشن یا برنامه ای که مرتب ورژن جدیدش میاد بیرون تا خطاهای قبلی خودش را برطرف کند و چیزهای جدیدی هم به آن اضافه شود و ارتقا پیدا کند. (این مدل برای افزایش خدمات و کاهش خطاها است.)

در این مدل ، ابتدا یک نمونه اولیه داریم که دائم در حال ارتقا است.

پس میتوان گفت این مدل قابلیت رشد دارد (**مشابه رشد انسان**) یعنی دائم در حال ارتقا میباشد.

مدل موازی (Parallel)



این مدل کاربرد کمتری دارد ولی در پروژه های بزرگ نیاز میشود.

در مدل موازی کارها به صورت موازی بین چندین گروه تقسیم میشوند که زمان انجام پروژه سریع تر شود که کار هر کدام متفاوت است.

از این مدل زمانی استفاده میکنیم که مثلا کارفرما میگه من میخوام پروژه و برنامم رو توی یک **زمان محدود** و زود درستش کنی. (مثلا میگه برنامم رو توی 3 ماه درست کن.)

پس از این مدل برای بالا بردن سرعت انجام پروژه و زمانی که حجم کار بالاست استفاده میشود.

در مدل موازی ، **از لحاظ هزینه هیچ محدودیتی وجود ندارد.** میتوانیم کار را به چندین گروه بسپاریم و گروه ها با مشورت یگدیگر کار را در بازه زمانی کم به اتمام می رسانند و در آخر محصولی که هر گروه تهیه کرده با محصول گروه های دیگر ادغام میشود و محصول نهایی ایجاد میشود.

تفاوت مدل موازی با مدل فازبندی شده :

در مدل فازبندی شده ، یک گروه تمام کار ها را انجام میداد ولی در مدل موازی کارها به صورت موازی بین چندین گروه تقسیم میشوند که زمان انجام پروژه سریع تر شود که کار هرکدام متفاوت است.

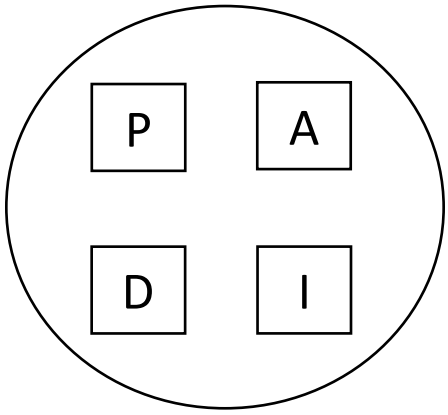
تفاوت مدل مارپیچی با مدل چرخشی :

در مدل چرخشی ، اون نمونه ی اولیه ای که تحویل کارفرما میدهیم ، نمونه ی بدرد بخور و قابل استفاده ای نیست ، و فقط برای این است که کارفرما یک شناخت و ذهنیتی از سیستم پیدا کند.

ولی در مدل مارپیچی ، اون نمونه ی اولیه ای که ما تحت عنوان "ورژن یک" تحویل کارفرما داده ایم ، قابل استفاده است و نصب و راه اندازی میشود.

نکته: برای استفاده از مدل موازی ، باید آن سیستم مورد نظر قابلیت موازی شدن را داشته باشد. چون برای بعضی از سیستم ها اصلا نمیتوان از مدل موازی استفاده کرد زیرا آن سیستم ها قابلیت موازی شدن را ندارند.

مدل چابک (Agile)



در این مدل **افراد حرفه ای** هستند و چند نفری کار میکنند و در این مدل اصول طراحی کمی نادیده گرفته میشود ، چون افراد حرفه ای هستند. (افراد با تجربه هستند.)

چون این افراد با تجربه هستند ، به جای اینکه طبق روش ها و مدل های گفته شده عمل کنند ، همه ی مراحل شناخت و تحلیل و طراحی و پیاده سازی رو همراه هم انجام میدهند.

کمی سیستم رو میسناسند ، کمی تحلیل اولیه میکنند ، کمی از سیستم رو طراحی میکنند و مقداری از سیستم رو به عنوان نمونه ، پیاده سازی میکنند و آن را به کارفرما تحویل میدهند. و دوباره همزمان با هم جلسه دارند و کار را با هم پیش میبرند. (این مدل برای افراد تازه کار نیست !!)

این افراد از قبل ، یک سری مستندات از قبل آماده شده دارند (چون قبلا کلی پروژه انجام دادند و حرفه ای و با تجربه شده اند.)

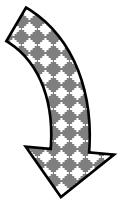
این افراد از کدهای قبلی استفاده شده ی خودشان استفاده میکنند . درکارهاشون خیلی مقید به مستند سازی نیستند.

ویژگی های مدل چابک:

1. گروه های حرفه ای
2. کمتر رعایت شدن اصول (مستند سازی / بایگانی کد ها)
3. کپی کردن کدها (از کدهای قبلی خودمان سعی میکنیم استفاده کنیم.)

در مقابل مدل چابک ، یک مدل دیگر به نام " USDP / UP " داریم.

که در آن اصول بیشتر رعایت میشود و به طور کامل مستند سازی داریم.



مدل (Unified Process) USDP / UP

Systems Development Life Cycle (SDLC)

" فرآیند توسعه نرم افزار یکپارچه "

که به آن " **فرآیند توسعه نرم افزار یکپارچه** " میگویند.

در این مدل ، برعکس مدل چابک ، اصول بیشتر رعایت میشود و به طور کامل مستند سازی داریم.

این مدل در زمینه ی کدنویسی سعی میکنه متکی به طراحی جدید باشد. و تحلیل و طراحی رو بر اساس اصول پیش میبرد. و برای پروژه های خیلی بزرگ از آن استفاده میکنیم.

نکته: ممکن ترین خطا در طراحی مربوط به زمان Design , Planning است.

ما در این درس ، بر مبنای این مدل و روش پیش میرویم تا بر اساس اصول و با دقت کارها را انجام دهیم و بعدا که کمی با تجربه شدیم ، میتوانیم از روش های دیگر هم استفاده کنیم.

نکته: در مدل چابک ، افراد مانند راننده ای هستند که خیلی در رانندگی حرفه ای است و بدون نگاه به دنده و ترمز و... به طور ناخودآگاه و براساس ملکه ی ذهنیش ، رانندگی میکند.

یک سوال مهم: در بررسی یک سیستم ، کدام یک از مدل ها بهترین مدل است ؟؟ یا درواقع کارای کدام بیشتر است ؟؟

جواب: نمیتوان گفت که کدام مدل برای همه ی سیستم ها بهترین است ، زیرا بستگی به شرایط و سیستم دارد که چه نوع سیستمی باشد و در مرحله دوم بستگی به کاربر دارد که چه چیزی میخواهد.

نکته: بدترین خطادر بررسی یک سیستم این است که ما در شناخت سیستم دچار خطا شویم و این خطا را تا مرحله ی پیاده سازی متوجه نشویم.

سوال: چطور ما میتوانیم تضمین دهیم که برداشتی که از سیستم موجود داشتیم ، یک شناخت و برداشت درستی است و نیازها را درست شناختیم ؟؟ یا شکستن به اجزای کوچکی که انجام دادیم کامل است ؟؟ (برای اینکه بدانیم سیستمی که ما در حال طراحی آن هستیم ، با سیستمی که کارفرما از ما میخواهد تطابق دارد یا نه)

ما میتوانیم با معیاری آن را مشخص کنیم.

معیاری به نام " چارچوب زکمن" / Zachman framework

این چارچوب به ما کمک میکنه تا نقاط کور و دیده نشده ی سیستم رو بشناسیم که در این چارچوب با سوال کردن ، شناختمون از سیستم بیشتر میشه. که این سوالات باید از دیدگاه افراد مختلف باشد. (از نگاه طراح – کارفرما – کاربر و...)

در صفحه ی بعد ، چارچوب زکمن را توضیح میدهیم.

بسم الله الرحمن الرحیم

نام و نام خانوادگی: حبیب صالحی **دانشگاه آزاد اسلامی واحد اهواز**

موضوع تحقیق: چارچوب زکمن

استاد مربوطه: دکتر مهدی مجد / سکشن: سه شنبه – 9/30 تا 12/15

تحلیل و طراحی سیستم ها – کلاس 119

Zachman Framework | (چارچوب زکمن)

چارچوب معماری زکمن که به نوعی جدول مندلیف مدل‌های معماری به حساب می‌آید. چارچوب زکمن به عنوان یک چارچوب مرجع در نظر گرفته می شود که شش جنبه اطلاعات، فرایندها، مکان ها، افراد، رویداد ها و اهداف را تحت پوشش قرار می دهد.

هدف چارچوب زکمن

هدف آن سازماندهی و تجزیه و تحلیل داده ها، حل مشکلات، برنامه ریزی برای آینده و ایجاد مدل های تحلیلی است.

با کمک این چارچوب ، نقاط کور رو میشناسیم و شناختمون از سیستم بیشتر میشه (با سوال پرسیدن)

که این سوالات باید از دیدگاه افراد مختلف انجام شود. (از نگاه طراح _کارفرما _ کاربر و...)

این چارچوب شش تحول مختلف را برای یک ایده انتزاعی (نه افزایش جزئیات، تنها تحول) را از شش دیدگاه مختلف فراهم می‌کند. این ویژگی اجازه می‌دهد تا افراد مختلف به یک جهت یکسان از دیدگاه‌های مختلف نگاه کنند. این کار باعث ایجاد **یک نگرش همگانی** از محیط می گردد.

ساختار آن

چارچوب زکمن از 36 دسته کلی برای توصیف هر چیزی از محصولات، خدمات، سخت افزار و نرم افزار استفاده می کند. دسته ها در شش ردیف و شش ستون سازماندهی شده اند و یک ماتریس دو بعدی با 36 سلول تشکیل می دهند که به شما کمک می کند **موضوع**، **مشکل** یا **محصول** را تجسم کنید.

ستون‌های یک الگوی چارچوب زکمن، سؤالات اساسی پیرامون معماری مورد نظر (چه کسی، چه چیزی، کجا و غیره) را ترسیم می‌کنند، در حالی که ردیف‌ها دیدگاه‌های هر نوع سهم و منفعت درگیر در پروژه را نشان می‌دهند. سپس ماتریس نهایی با فرآیندها، اقلام مورد نیاز لازم، نقش‌های مهم، مکان‌های مرتبط و هر هدف یا قانون مرتبط با پروژه، بر اساس سؤال اساسی و دیدگاهی که در هر سلول نشان داده شده است، پر می‌شود.

شش ردیف ماتریس چارچوب زکمن عبارتند از:

۱. **دیدگاه برنامه ریز (محدوده)** (Planner's view): این ردیف جایی است که شما طرح یا استراتژی کسب و کار را شناسایی می کنید و مشخص می کنید که چه موضوع یا نگرانی در ماتریس مورد توجه قرار می گیرد.

۲. **دیدگاه مالک (مفاهیم تجاری)** (Owner's view): ردیف دوم جایی است که نیازهای کسب و کار و منابعی را که کسب و کار برای اجرای طرح به آن نیاز دارد، شناسایی می کنید.

۳. **دیدگاه طراح (منطق سیستم)** (Designer's view): ردیف سوم مشخص می کند که چگونه **طرح کسب و کار** نیازهای کسب و کار را برآورده می کند. این ردیف مربوط به کار انجام شده توسط تحلیلگران سیستم است که داده ها، جریان های فرآیندی و عملکردهای فرآیندهای تجاری را مدیریت می کنند.

۴. **دیدگاه مهندس (فیزیک فناوری)** (Engineer's perspective): ردیف چهارم شامل اطلاعات مرتبط در مورد نحوه اجرای استراتژی و ابزارها، فناوری، اقلام مورد نیاز و محدودیت‌هایی است که تیم با آنها کار خواهد کرد.

۵. **دیدگاه تکنسین (مونتاژ اجزا) (Technician's perspective)**: این ردیف جایی است که شما نمایی از نیازهای محصولات، خدمات یا سخت افزار را در آن قرار می دهید.

۶. **دیدگاه کاربر (کلاس های عملیات) (User's view)**: ردیف آخر شامل اطلاعاتی در مورد عملکرد سیستم و نحوه عملکرد آن در محیط فناوری اطلاعات یا کسب و کار است.

شش ستون ماتریس زکمن شامل تمام سوالاتی است که در طول فرآیند از خود می‌پرسید:

۱. **چه (داده) (what)**: این جایی است که شما تعیین می کنید که چه داده ها، اطلاعات و الزامات تجاری برای پروژه ضروری است.

۲. **چگونه (عملکرد) (How)**: ستون "چگونه" یا "عملکرد" نحوه عملکرد فرآیندها و تأثیرگذاری بر تجارت را مشخص می کند.

۳. **کجا (شبکه) (where)**: این ستون اشاره به مکان ها دارد که شامل تمام شبکه‌های سیستم و مکان‌های مربوطه می‌شود که در آن عملیات تجاری انجام می‌شود.

۴. **چه کسی (افراد) (Who)**: در ستون چهار، ذینفعان کلیدی را شناسایی کرده و همه پرسنل مربوطه را برای پروژه تعیین می کنید.

۵. **چه زمانی (زمان) (When)**: ستون پنج جایی است که شما زمان و زمان انجام فرآیندهای کسب و کاری در شرکت را مشخص می کنید.

۶. **چرا (انگیزه) (Why)**: ستون پایانی جایی است که شما مشخص می کنید که چرا راه حل نهایی را انتخاب کرده اید و انگیزه پشت اقدام یا پروژه چیست.

	داده ها (چه)	کارکرد (چگونه)	شبکه (کجا)	مردم (کی)	زمان (کی)	انگیزه (چرا)
هدف/حوزه مضمونی نقش: برنامه ریز	فهرست چیزهای مهم برای حرفه	فهرست فرایندهای هسته ای حرفه	فهرست مکانهای حرفه	فهرست سازمانهای مهم	فهرست رخدادها	فهرست اهداف/ استراتژیهای حرفه
مدل سازمانی مفهومی نقش: صاحب	مدل شئ/دادهٔ مفهومی	مدل فرایندی حرفه	سیستم لجستیکی حرفه	مدل جریان کار	زمانبندی اصلی	طرح حرفه
مدل سیستمی منطقی نقش: طراح	مدل منطقی داده ها	مدل معماری سیستم	معماری سیستمهای توزیع شده	معماری واسط انسانی	ساختار پردازشی	مدل نقش حرفه
مدل فناوری فیزیکی نقش: سازنده	مدل داده/کلاس فیزیکی	مدل طراحی فناوری	معماری فناوری	معماری نمایش	ساختار کنترلی	طراحی قاعده
نمایشهای جزئی خارج از مضمون نقش: برنامه نویس	تعاریف داده ها	برنامه	معماری شبکه	معماری امنیتی	تعریف زمانی	توصیف قاعده
سازمان در حال کار نقش: کاربر	دادهٔ قابل استفاده	نایع در حال کار	شبکهٔ قابل استفاده	سازمان در حال کار	زمانبندی پیاده سازی شده	استراتژی در حال کار

چارچوب زکمن هفت قانون یا اصل راهنما را برای تکمیل ماتریس دو بعدی خود ایجاد کرده است:

1. ستون ها ترتیبی ندارند، اما باید به ترتیب از بالا به پایین مرتب شوند که از مهم ترین دسته شروع می شود. اولویت ستون ها وابسته به پروژه یا نگرانی فناوری اطلاعات شما خواهد بود و ممکن است زمانی که برای محصول یا خدمات دیگری اعمال شود تغییر کند. شما باید از افزودن یا حذف هر ستون یا ردیف خودداری کنید، زیرا برای به دست آوردن تصویر کامل به همه آنها نیاز دارید.

2. هر ستون یک مدل عمومی ساده دارد و می تواند متا مدل خود را در آن ستون داشته باشد.

3. مدل اصلی هر ستون باید منحصر به فرد باشد و از همپوشانی یا تکرار داده ها در ستونهای دیگری اجتناب کند.

4. هر ردیف یک چشم انداز متمایز و منحصر به فرد را توصیف می کند. شما باید از داشتن هر گونه متا مدل یا مفهومی که به سلول های متعدد نسبت داده می شود خودداری کنید. یک عنصر کلیدی این چارچوب این است که از همه تکرار ها در ماتریس دو بعدی نهایی جلوگیری می کند.

5. اگر در قوانین 2، 3 و 4 موفق هستید، باید ماتریسی داشته باشید که در آن هر سلول منحصر به فرد است. این به شدت تاکید شده است و یکی از سنگ بناهای این چارچوب است که در نتیجه نمای منحصر به فرد و آموزنده ای از معماری شما ایجاد می کند.

6. از تغییر نام سطرها یا ستون های خود خودداری کنید. اگر سهم و منفعت از اصطلاحات مشابه به طور متفاوت استفاده کنند، این می تواند معنی را تغییر دهد یا باعث سردرگمی شود.

7. منطق بازگشتی و عمومی است، به این معنی که می توان از آن برای طبقه بندی یا تجزیه و تحلیل هر چیزی که مربوط به معماری سازمانی مورد نظر است استفاده کرد. تعیین هدف و مرزها به عهده تحلیلگر است و این تصمیمات می توانند تأثیر قابل توجهی بر نتیجه نهایی ماتریس و ابتکار یا پروژه داشته باشند.

هر سطر نشان دهنده یک دید کلی از راه حل از یک دیدگاه خاص است. یک سطر یا دیدگاه بالاتر لزوماً نباید نسبت به دیدگاه پایین‌تر درک جامع تری از کل مسئله داشته باشد. هر سطر نشان دهنده یک دیدگاه متمایز و منحصر به فرد است، با این حال، خروجی‌ها از هر دیدگاه باید جزئیات کافی برای تعریف راه حل در سطح خود دیدگاه را فراهم کنند و همچنین باید برای استفاده در دیدگاه سطر پایین‌تر به صورت صریح ترجمه کردند.

هر دیدگاه باید نیازها و محدودیت‌ها و الزامات دیگر دیدگاه‌ها را مد نظر قرار دهد. محدودیت‌های دیدگاه‌ها به صورت افزودنی هستند. برای مثال محدودیت‌های سطر بالاتر سطر پایین‌تر را تحت تأثیر قرار می‌دهد. محدودیت‌های سطر پایین‌تر می‌تواند، اما نه لزوماً، بر سطرهای بالاتر تأثیر بگذارد. درک الزامات و محدودیتها مستلزم تبادل دانش و داشتن درک متقابل از دیدگاهی به دیدگاه دیگر است. چارچوب از مسیر عمودی برای ارتباطات بین دیدگاه‌ها استفاده می‌کند.

پایان

SDLC System Development Life Cycle / چرخه ی حیات توسعه نرم افزار

چرخه ی حیات توسعه ی نرم افزار ، مجموعه‌ای از مراحل است که برای ایجاد برنامه‌های نرم‌افزاری استفاده می‌شود. این مراحل، فرایند توسعه را به وظایفی تقسیم می‌کنند که سپس می‌توان آن‌ها را تعیین، تکمیل و اندازه‌گیری کرد.

چرخه حیات توسعه نرم‌افزار، کاربرد روش‌های استاندارد کسب و کار در ساخت برنامه‌های نرم‌افزاری است. این برنامه به طور معمول به شش تا هشت مرحله تقسیم می‌شود: برنامه‌ریزی، نیازمندی‌ها، طراحی، ساخت، مستندسازی، آزمایش، استقرار، نگهداری.

SDLC مشخص میکنه که چه کارهایی را انجام بدهیم.

چند نمونه از کارهای جزئی در SDLC (کارهایی که باید انجام بدهیم) :

Planning	Analysis	Design	Implementation
تحلیل امکان سنجی (Feasibility Analysis)	تعیین استراتژی (Analysis Strategy)	تعیین استراتژی طراحی (Design Strategy)	ساخت و ساز (Construetion)
مدیریت پروژه (Project Management)	جمع آوری نیازها (Requirement Gathering)	تعیین ساختار معماری (Architecture Design)	نصب و راه اندازی (Installation)
	پیشنهاد سیستم (System proposal)	طراحی برنامه (Program Design)	طراح حمایت و/ برنامه ریزی پشتیبانی (Support Plan)

امکان سنجی

امکان سنجی یا **مطالعات امکان سنجی**، ارزیابی و تجزیه و تحلیل پتانسیل یک پروژه پیشنهادی است و بر اساس تحقیقات و مطالعاتی پایه ریزی شده‌است که روند تصمیم‌گیری را پشتیبانی کند.

یک مطالعه امکان سنجی، ارزیابی عملی بودن یک پروژه یا سیستم پیشنهادی است.

امکان پذیری یک کار را بررسی میکنیم.

انواع امکان سنجی:

1. امکان سنجی تکنیکی
2. امکان سنجی مالی
3. امکان سنجی سازمانی
4. امکان سنجی فنی

و...

Model : یک ساده سازی از واقعیت است / که **هدف آن** ایجاد ارتباط استاندارد بین طرفین (گروه طراح و کارفرما) و پیدا کردن یک درک مشترک ، ساده سازی و مستند سازی استاندارد است. و درواقع از آن به عنوان یک الگو استفاده میکنیم و مستند سازی بر اساس همین مدل ، استاندارد میشود.

در ایجاد یک مدل ، از جزییات صرف نظر میکنیم و درگیر جزییات نامربوط نمیشویم.

نکته: (منظور از "واقعیت" همان سیستم فعلی است ، پس درواقع مدل یک ساده سازی از سیستم فعلی است.)

مدلی که ما میخواهیم از سیستم خود برداشت کنیم ، طبق تجربه ی خودمون به وجود می آید. و در این مرحله هنوز روش های مهندسی شده ی موجود شده وجود ندارند.

و درواقع مدل ها میتواند بر اساس علایق و سلایق هر شخص ، با دیگران متفاوت باشد.

و مدل های اشخاص و هر شرکت با شرکت دیگر ، متفاوت و مختص به خودش است.

که همین امر باعث شد در حدود دهه ی 90 میلادی (1990) ، **زبان های مدل سازی مختلفی** به وجود آیند. که شرکت ها بر اساس آن روش ها ، مدل خودشان را طراحی میکردند.

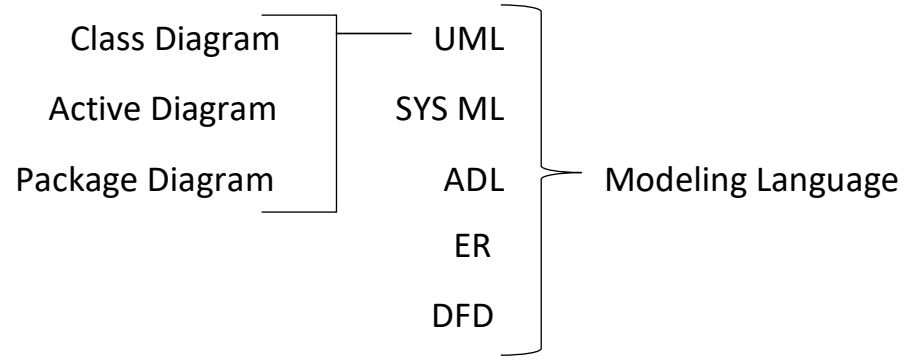
Modeling Language (زبان های مدل سازی) / (ML) : یک استانداری برای دریافت از کارفرما ، انجام تحلیل و طراحی و تحویل محصول است.

مجموعه ای از ابزار را معرفی میکند که به ما کمک میکند مدل هایی بکشیم که شامل دیاگرام هایی باشد که برای ارتباط بین مهندسی نرم افزار استفاده میشوند.

زبان مدل سازی مانند یک جعبه ابزار است که نحوه ی انجام کار را نمیگوید ، و "متدولوژی" است که نحوه ی انجام کار را میگوید. (متدولوژی = جعبه ابزار + تکنیک ها)

در مدل‌سازی علمی و مهندسی سیستم‌ها و نرم‌افزار، به هر زبان ساختگی اطلاق می‌شود، که قابلیت بیان اطلاعات و دانش یا معرفی سیستم‌ها را دارا باشد و در یک ساختار تعریف شده، توسط مجموعه‌ای از قوانین (به منظور تفسیر اجزای ساختار) مورد استفاده قرار گیرد.

زبان های مدل سازی انواع مختلفی دارند که عبارت اند از:



Modeling Language

این مدلینگ میتونه بر اساس ساختار گرافیکی مدل سازی بشه.(Graphical)

و یا میتواند دیاگرام داشته باشد.(Diagram)

تاکید بر زبان هم میتواند شامل یک سری قواعد و گرامرهایی باشد. باد و نباید هایی که ما برای انجام کار رعایت میکنیم. پس گرامرهایی را که درقالب گرافیکی و دیاگرامی رعایت میکنیم را درنظر داریم.

زبان مدل سازی مانند یک جعبه ابزار است که نحوه ی انجام کار را نمیگوید ، و "متدولوژی" است که نحوه ی انجام کار را میگوید. (متدولوژی = جعبه ابزار + تکنیک ها)

جعبه ابزار UML :

(نمودار فعالیت) / روند برنامه را مدلسازی میکند. /

1) Activity Diagram

Activity Diagram: مجموعه ای از Probability Event (رویداد احتمالی) مثل فلوچارت است.

- ✓ فعالیت عملی است که به وسیله انسان یا کامپیوتر انجام می‌شود و در دیدگاه پیاده سازی، متد است.
- ✓ نمودار فعالیت یک فلوچارت است که برای نمایش جریان کنترل از یک فعالیت به فعالیت دیگر به کار می رود.
- ✓ از بسیاری جهات شبیه state diagram است.
- ✓ نمودار فعالیت برای نمایش جریان کار و نمایش رفتاری که پردازش‌های موازی دارند مناسب است.
- ✓ در این نمودار صرف نظر از اینکه فاعل رفتار چه کسی است، میتوان رفتار را به خوبی با تقدم و تاخّر و بیان شرط های لازم نمایش داد.

Use Case: کارهایی هستند که ما باید مرحله به مرحله بین کاربر و سیستم برای رسیدن به هدف انجام بدیم. (در دنیای نرم افزاری است. / و بخشی از فرآیند سازمانی

2) State Diagram (نمودار حالت) / مدل کردن حالت ها /

3) Use Case Diagram / مدل کردن کارهای سیستم ← Use Case

معرفی Use Case

Use Case (یوزکیس) ابزاری برای تعریف تعاملات مورد نیاز کاربر در سیستم است، در واقع مجموعه اقداماتی است که مرحله به مرحله تعاملات بین کاربر و سیستم را برای رسیدن به یک هدف خاص (که همان کامل شدن کیس است) تعریف می کند. کیس را می توان یک کار در نظر گرفت که باید تکمیل شود.

در توسعه نرم افزار Use Cases یا مجموعه ای از Caseها (کیس ها) نوشته می شود. در تصویر زیر هر کدام از بیضی ها یک کیس را نشان می دهد و کاربری که با سیستم از طریق این کیس ها ارتباط برقرار می کند.

یک یوزکیس دیاگرام عمدتا شامل اکتور یا کنشگر (actor)، مورد کاربرد (use case) و ارتباطات (relationships) است. دیاگرام های پیچیده و بزرگتر شامل سیستم ها (systems) و مرزها (boundaries) هم می شوند. ما در این مطلب یوزکیس دیاگرام مبتنی بر شئ (object) را مورد بررسی قرار می دهیم.

چه کسانی از مستندات "Use Case" استفاده می کنند؟

مستندات Use Case یک نمای کامل از مسیرهای مختلف تعاملات کاربر با سیستم برای رسیدن به هدف ارائه می دهد. هرچه اسناد بهتری نوشته شود، شناسایی نیازمندی های یک سیستم نرم افزاری بسیار ساده تر خواهد شد.

توسعه دهندگان نرم افزار، آزمایش کنندگان نرم افزار و همچنین سایر گروه های ذی نفع می توانند از این مستندات استفاده کنند.

موارد استفاده از مستندات Use Case:

- توسعه دهندگان برای پیاده سازی کد و طراحی آن
- آزمایش کنندگان برای ایجاد کیس های آزمایشی
- ذی نفعان تجاری برای درک نیازهای نرم افزار

کنشگرها (Actors)

- **اکتورها یا کنشگرها شروع کننده یک فعالیت در سیستم مورد نظر هستند**– شما در ابتدا باید کنشگرهای مرتبط با کسب و کار مد نظرتان را نامگذاری کنید. به عنوان مثال اگر یوزکیس شما با یک سازمان خارجی تعامل دارد بهتر است به جای استفاده از نام آن سازمان برای یوزکیس، عملکرد آن سازمان را به عنوان اسم به کار ببرید (مثلا Airline Company بهتر از PanAir است).
- **اکتور یا کنشگرهای اولیه (Primary Actors) باید در سمت چپ نمودار شما قرار بگیرند**– این کار باعث می شود تا به سرعت بتوانید نقش های مهم و کلیدی موجود در سیستم را برجسته کنید.
- **نقش کنشگرها (نه موقعیت هایشان) را عنوان کنید-** به عنوان مثال، در یک هتل هم مدیر دفتر و هم مدیر شیفت می توانند کار رزرو کردن را انجام بدهند. بنابراین با استفاده از نامی مثل "مأمور رزرو" باید نقش هر کنشگر در سیستم را مشخص نمایید.
- **سیستم های خارجی کنشگر هستند-** اگر یوزکیس شما ارسال ایمیل است و با نرم افزار مدیریت ایمیل تعامل دارید، این نرم افزار یک کنشگر برای یوزکیس خاص شما محسوب می شود.
- **کنشگرها با یکدیگر تعامل ندارند-** در صورتی که کنشگرهای یک سیستم با هم تعامل دارند، باید یک یوزکیس دیاگرام جدید ایجاد کنید که سیستم ارائه شده در یوزکیس دیاگرام قبلی را به عنوان یک کنشگر نشان بدهد.
- **کنشگرهای ارث بری شده (inheriting actors) را زیر کنشگرهای والد (parent actors) قرار بدهید**– این کار برای خوانایی بیشتر و برجسته کردن سریع موارد کاربرد خاص برای هر کنشگر انجام می شود.

یوزکیس ها (Use Cases)

- **نام یوزکیس ها با یک فعل شروع می شود-** یک یوزکیس بیان کننده عمل یک مدل یا سیستم است، بنابراین اسم آن باید با یک فعل شروع شود.
- **نام یوزکیس را توصیفی انتخاب کنید-** توصیفی کردن اسم یوزکیس باعث می شود افرادی که به نمودار شما نگاه می کنند اطلاعات بیشتری از آن دریافت کنند. به عنوان مثال نام "چاپ فاکتور" بهتر از "چاپ" است.
- **ترتیب منطقی یوزکیس ها را برجسته کنید-** برای مثال، اگر در حال تجزیه و تحلیل اطلاعات برای یک مشتری بانک هستید، یوزکیس های معمول شما شامل باز کردن حساب، سپرده گذاری و برداشت می شود. موقع ترتیب بندی این موارد، آنها را به شکلی معقول و منطقی بچینید.
- **یوزکیس های اینکلود یا شامل شده (included use cases) را در سمت راست یوزکیس قرار بدهید-** این کار برای بهبود خوانایی و افزایش وضوح انجام می شود.
- **یوزکیس های ارث بری شده (inheriting use case) را زیر یوزکیس های منبع یا والد (parent use case) قرار بدهید-** این کار هم برای افزایش خوانایی و فهم نمودار شما انجام می شود.

4) Class Diagram / نمودار کلاس

Case Tools
- Rational Rose
- Enterprise Architect
- Visio
- visual paradigm

بعضی افراد طی ابدا تکنیک های نرم افزاری و برنامه نویسی ، نگاه های مختلفی رو نسبت به سیستم ها پیدا میکنند که یکی از این نگاه ها ، نگاه های داده محور است که در آن سیستم را به داده هایش میبینند. اصل هر موجودیتی که در سیستم هست را داده میگیرند.

به عنوان مثال دانشجویی که در دانشگاه ثبت نام میکند با مشخصات داده ای خودش نمو پیدا میکند. مثل اسم ، فامیل ، اطلاعات شناسنامه ای ، شماره دانشجویی ، تعداد و عناوین درس های پاس شده ، لیست دروس انتخاب واحدش ، معدل و...

(پس موجودیت آن ، با داده هایش نمو پیدا میکنه **!!!!** و مولف هر موجودتی را داده هایش در نظر میگیرند.)

انواع دیدگاه ها در تحلیل و طراحی سیستم ها:

- Data Center(داده محور)
- Procedural / Functional
- Object Oriented(شی گرا)

متدولوژی (Methodology) **:**

نگاه ما را به سیستم بیان میکند. اینکه نگاه ما در تحلیل و طراحی شامل چه چیزی است. وسیله شناخت هر علم است. روش‌شناسی در مفهوم مطلق خود به روش‌هایی گفته می‌شود که برای رسیدن به شناخت علمی از آن‌ها استفاده می‌شود.

Methodology :

Process Model + Modeling Language + تکنیک های تحلیل و طراحی

+ Implementation Framework

در انتخاب یک متدولوژی ، موارد زیر را بررسی میکنیم.

+ Process Model

+ SDLC (چرخه حیات نرم افزار)

+ Modeling Language

+ تکنیک هایی برای توسعه

از یک زمانی به بعد ، شرکت ها و محققین به این فکر افتادند که یک **زبان مدلسازی یکپارچه** طراحی کنند که یک روش استانداری باشد ، پس یک زبان مشترک طراحی کردند.

Unified Modeling Language / UML

(زبان مدلسازی یکپارچه)

روش ها و دیدگاه های مختلف در نوشتن زبان های مدلسازی و برنامه نویسی و سیستم های نرم افزاری:

- 1) **Code & Fix**: در مراحل اولیه ، یعنی همان دهه 80 میلادی ، کدنویسی ها به شکل "کد & فیکس" بوده. که در آن اول کد نوشته میشود و بعد رفع خطا شروع میشود ، که همین کار که رفع خطا بعد از انجام کد نویسی باشد و در حین انجام کدنویسی نباشد ، باعث همان بحران بزرگ شد که در صفحات قبل به آن پرداختیم.

بعد از آن افراد به این فکر و ایده افتادند که یک مقدار بیشتر نگاه تجزیه و تحلیلی بهتری را داشته باشند.

که در این نگاه تجزیه و تحلیلی ابتدا داده ها را نگاه میکردند. یعنی اینکه موجودیت های داخل سیستم با داده هایشان نمو پیدا میکنند. و از همینجا بود که نگاه های Data Center

(یعنی نگاه داده محور / مبتنی بر داده) به وجود آمد و شکل گرفت.

که در آن هرچیزی را به داده ها میبینند و بقیه ی کارها از اون داده ها گرفته میشوند.

- 2) **Data Center**:

بعضی افراد طی ابدا تکنیک های نرم افزاری و برنامه نویسی ، نگاه های مختلفی رو نسبت به سیستم ها پیدا میکنند که یکی از این نگاه ها ، **نگاه های داده محور** است که در آن سیستم را به داده هایش میبینند. اصل هر موجودیتی که در سیستم هست را داده میگیرند. (نگاه داده محور /Data Center)

به عنوان مثال دانشجویی که در دانشگاه ثبت نام میکند با مشخصات داده ای خودش نمو پیدا میکند. مثل اسم ، فامیل ، اطلاعات شناسنامه ای ، شماره دانشجویی ، تعداد و عناوین درس های پاس شده ، لیست دروس انتخاب واحدش ، معدل و...

(پس موجودیت آن ، با داده هایش نمو پیدا میکنه !!!! و مولف هر موجودتی را داده هایش در نظر میگیرند.)

بعد از آن به این نتیجه رسیدند که بهتر است برنامه نویسی را "ساخت یافته" کنند.

و برنامه نویسی ساخت یافته را به وجود آوردند.

3- Behavioral / Procedural / Functional / **Structured**

توضیحات "ساخت یافته" زیر ، از سایت ویکی پدیا در این جزوه آمده است تا توضیحات دقیقی ارائه شود.

برنامه‌نویسی ساخت‌یافته (به انگلیسی: Structured programming) یک پارادایم برنامه‌نویسی است که طبق آن برنامه‌نویس قدم‌ها و روال‌هایی را که لازم است تا برنامه به جواب برسد را مشخص می‌کند. در این روش از برنامه‌نویسی، انجام یک روال به روال‌های کوچک‌تر تقسیم می‌شود و به این ترتیب یک برنامه با شکسته شدن به ریز برنامه‌های کوچک‌تر سعی می‌کند تا عملکرد مد نظر را پیاده‌سازی کند.

رویه‌ها (به انگلیسی: routines)، زیر رویه‌ها (به انگلیسی: subroutines)، ساختار بلوک (به انگلیسی: block structures) و حلقه‌های for و while در کنار سادگی آزمودن کدها و صرف نظر کردن از Goto که برنامه را به یک کلاف سردرگم (به اصطلاح برنامه‌نویسی: spaghetti code) تبدیل می‌کرد، موجب شدند تا دنبال کردن برنامه و نگهداری از آن تا حد زیادی بهبود یابد.

این پارادایم در دههٔ ۱۹۶۰ توسط بوهن (به انگلیسی: Böhm) و جاکوپینی (به انگلیسی: Jacopini) پدید آمد و در سال ۱۹۶۸ پدیدهٔ معروفی به نام Goto از سوی ادسخر دیکسترا زیان‌آور تشخیص داده شد و این پدیدهٔ تازه به صورت تئوری در قالب برنامه‌نویسی ساخت‌یافته ارائه شد و پس از آن توسط زبان الگول (به انگلیسی: ALGOL) به کمک ساختارهای کنترلی پشتیبانی گردید.[۱][۲]

مثال [ویرایش]

به عنوان مثال برای نوشتن برنامه‌ای که قراراست اطلاعات نمرات یک محصل را بگیرد و کارنامهٔ آن را چاپ کند، زیر روال‌های زیر لازم است:

- زیر روالی برای خواندن اطلاعات ورودی
- زیر روالی برای جمع‌آوری اطلاعات ورودی و محاسبهٔ معدل
- زیر روالی برای چاپ اطلاعات به صورت یک جدول
- زیر روالی برای اتصال به چاپگر و چاپ گزارش

هر زیر روال آنقدر کوچک می‌شود که برنامه‌نویس بتواند راحت‌تر کار کردن آن را درک کند (هر زیر روال معمولاً ۳۰ خط برنامه‌نویسی است). به این ترتیب برنامه‌نویس با نوشتن هر زیر روال بخشی از برنامه را تولید می‌کند و برنامه‌نویسان مختلف می‌توانند بر روی زیر روال‌های مختلف کار کنند تا در نهایت به اضافه نمودن آن‌ها به یکدیگر برنامهٔ نهایی ساخته شود.

در زبان‌های ساختار یافته توابع کتابخانه‌ای فراوانی وجود دارند که سعی می‌کنند به برنامه‌نویس در برخی از روال‌ها کمک کنند؛ مثلاً برای چاپ در مثال فوق، توابع کتابخانه‌ای برای سهولت انجام کار در این زیر روال، در زبان پاسکال، وجود دارد.

بعد از آن یک نگاه تحولی به وجود آمد که ابتدا در برنامه نویسی شروع شد و بعدا در بقیه جاها ، اون اثر خودش رو نمایان کرد.

که از کنار هم قرار گرفتن داده ها و عملکرد ها ، شئ به وجود آمد.

داده + عملکر وروال ها = شئ

که در نتیجه نگاهی به نام Object Oriented **(شی گرا) به وجود آمد.**

4- Object Oriented

این نگاه ابتدا در برنامه نویسی ابدا شد و هم بعدا در تحلیل و طراحی سیستم و... خودش رو نمایان کرد.

برنامه نویسی شئ گرا چیست ؟

در اولین زبان‌های برنامه‌نویسی تمام برنامه را در کدهای پشتِ سرِ هم می‌نوشتیم. بعد از آن، شیوه‌های رویه‌ای روی کار آمدند که کدها را به قطعه‌های مختلفی که هر کدام عملیات خاصی انجام می‌دهند تقسیم می‌کنیم. سپس هر کجا که نیاز به اجرای آن‌ها داریم، یک تابع برنامه‌نویسی را فراخوانی می‌کنیم.

اگر بخواهیم یک پروژه طبق نیازمندی‌های دنیای واقعی انجام دهیم، کدهای بسیار طولانی خواهیم داشت. اجازه دهید با یک مثال ساده توضیحاتم را ادامه دهم.

فرض کنید می‌خواهیم یک سیستم مدیریت دانشگاه پیاده‌سازی کنیم. دانشجویان، اساتید و کارمندان دانشگاه بخشی از این سیستم خواهند بود. احتمالاً قبول دارید که این سه نوع داده، همگی انسان هستند اما هر کدام ویژگی‌ها یا دسترسی‌های متفاوتی در سیستم دارند.

در حالت رویه‌ای احتمالاً سه نوع داده تعریف می‌کنیم که کاملاً از یکدیگر جدا هستند. حال اگر بخواهیم تغییر کوچکی در یکی از ویژگی‌های مشترک این‌ها اعمال کنیم، مجبوریم این تغییر را ۳ بار تکرار کنیم !!

در ادامه متوجه می‌شوید که پیاده‌سازی و اعمال تغییرات در سیستم (قابلیت نگهداری) با برنامه‌نویسی شیءگرا چقدر آسان‌تر می‌شود.

مفاهیم شیء گرایی

چهار مفهوم یا کلمه در برنامه نویسی شئ گرا به دفعات استفاده می‌شود. پس بهتر است همین ابتدا در مورد این ۴ تا به توافق برسیم. این مفاهیم عبارت‌اند از:

۱. **کلاس** (**Class**)

۲. **شئ** (**Object**)

۳. **ویژگی** (خصیصه یا **Property**) که گاهی به آن شناسه (**Attribute**) گفته می‌شود.

۴. **رفتار** (**Behavior**) که به آن **متد** (**Method**) هم می‌گوییم.

به زبان خیلی ساده، کلاس یک الگوی (نقشه) کلی برای نوع داده‌ای ما در سیستم است. هر کلاس دارای ویژگی‌ها و رفتارهایی است که برای تمام داده‌های از آن نوع قابل تعریف است. هر گاه از روی این الگو یک داده ساخته و ویژگی‌های آن را تعیین کنیم، یک شئ از آن کلاس در اختیار داریم.

به عملیات ساخت شئ از کلاس، نمونه‌سازی (**ساخت instance**) گفته می‌شود.

مثال کلاس و اشیاء در OOP

کلاس، متد و ویژگی در برنامه‌نویسی

در مثال سیستم دانشگاه، یک نوع داده‌ای خیلی جامع به اسم «انسان» داریم. مفهوم انسان در دنیای واقعی را تصور کنید. این موجود دارای یک ساختار کلی است که از مجموعه‌ای از ویژگی‌ها و رفتارها ایجاد شده‌اند. برای مثال:

- ☑ سن، رنگ پوست، قد و وزن برخی از ویژگی‌های انسان هستند.
- ☑ راه رفتن، دیدن، خوردن، شنیدن و حرف زدن از رفتارهای اوست. این‌ها رفتارها (متدها) شبیه تابع‌هایی هستند که روی اشیاء از نوع این کلاس قابل اجرا (اصطلاحاً صدا زدن) هستند.

> هر گاه مطابق الگوی «انسان» (کلاس) ویژگی‌های خاصی تعریف کنیم، از آن یک شئ ساخته شده است. همه شئ‌های این کلاس، دارای رفتارهای مشابهی هستند اما به دلیل تفاوت در ویژگی‌هایشان تا حد زیادی از یکدیگر متمایز خواهند بود.

آیا می‌توانید یک نوع حیوان (مثلاً گربه) را به صورت کلاس مدل‌سازی کنید؟ سعی کنید این کار را انجام دهید و اگر سؤالی داشتید از قسمت دیدگاه‌های آموزش مطرح کنید.

اصول شیءگرایی به زبان ساده

تا این جا با مفاهیم یا بهتر بگویم، اجزای یک برنامه بر اساس برنامه نویسی شئ گرا آشنا شدیم. مثال‌های بیشتر را در ادامه می‌زنم که این تعاریف را کامل درک کنید.

برنامه‌نویسی شئ‌گرا دارای اصول خاصی است. یعنی علاوه‌بر اینکه باید تمام موجودیت‌های برنامه را بر اساس کلاس‌ها تعریف کنیم، لازم است اصول خاصی را رعایت کنیم.

البته اگر این اصول را رعایت نکنیم یا به درستی از آن‌ها استفاده نکنیم، برنامه ما همچنان یک برنامه شئ‌گرا (Object Oriented) خواهد بود، ولی کیفیت لازم را نخواهد داشت. از این اصول در نوشتن برنامه‌ها بسیار استفاده می‌شود و درک و فهم آن‌ها به شما کمک زیادی خواهد کرد.

تجرید یا انتزاع در برنامه‌نویسی

انتزاع (Abstract) به معنی محدود کردن جزئیات برای پرداختن به فرآیند اصلی است. اجازه دهید با دو مثال از زندگی روزمره شروع کنم:

- ☑ خیلی از ما به عنوان راننده پشت یک ماشین نشسته‌ایم و آن را روشن کرده‌ایم، اما همه از ساز و کارت استارت خوردن و نحوه کار دینام‌ها اطلاعی نداریم.
- ☑ تا به حال با ماکروویو، قهوه ساز یا لباس‌شویی کار کرده‌ایم ولی از ساز و کار آن‌ها اطلاع دقیقی نداریم.

ما در نهایت انتظار داریم کاری که می‌خواهیم به درستی انجام شود و به اینکه این کار چطوری انجام می‌شود هیچ کاری نداریم! معنای انتزاع همین است. ما به عنوان مصرف‌کننده یک وسیله، نیازی نداریم بدانیم پس از فشرده شدن یک کلیک، چه فعل و انفعالاتی صورت می‌گیرد تا نتیجه حاصل شود. برای ما فقط نتیجه مهم است.

در هنگام تعریف رفتارها (متدها) و ویژگی‌ها در برنامه‌نویسی شئ‌گرا باید به این مسئله دقت کنیم که مصرف‌کننده این کلاس به چه سطح از تجرید نیاز دارد. بهتر است چه بخش‌هایی از الگوریتم را از دید مصرف‌کننده پنهان کنیم؟

> توجه داشته باشید که از انتزاع برای رسیدن به سادگی در استفاده کمک می‌گیریم. یعنی هدف ما راحت‌تر شدن کار با کلاس و متدهاست.

کپسوله‌سازی در برنامه‌نویسی شیءگرا

مطمئناً هر برنامه از اشیاء مختلفی تشکیل شده است. **اصل کپسوله‌سازی (Encapsulation)** به ما می‌گوید که هر شئ باید از ویژگی‌های خود محافظت کند. یعنی بهتر است اشیاء دیگر به طور مستقیم به ویژگی اشیاء دیگر دسترسی نداشته باشد.

کپسوله سازی یا Encapsulation برای اطمینان و امنیت بیشتر

فرض کنید می‌خواهیم یک بازی شطرنج را به صورت شئ‌گرا پیاده‌سازی کنیم. بازی شطرنج از مهره‌هایی تشکیل شدند که هر کدام دارای ویژگی‌ها و رفتارهای مختلفی هستند. یعنی:

- ☑ هر مهره دارای رنگ، شکل، اسم (نوع مهره) و محل قرارگیری خاصی روی صفحه بازی است.
- ☑ هر مهره با توجه به نوعش می‌تواند به شیوه مختلفی روی صفحه بازی حرکت کند.

در یک پیاده‌سازی خیلی ساده، محل مهره را با یک متغیر (ویژگی در کلاس) از نوع عدد مشخص می‌کنیم. همچنین چون همه مهره‌ها دارای رفتارها و ویژگی‌های یکسانی هستند، یک کلاس اصلی برای مهره بازی در نظر می‌گیریم.

شاید بگویید همه مهره‌ها رفتارهای مشابهی ندارند. چون حرکت اسب در صفحه بازی با حرکت رخ بسیار متفاوت است. به کمک ۲ اصل بعدی در برنامه نویسی شئ گرا می‌توانیم این تفاوت‌های جزئی را پیاده‌سازی کنیم.

فعلاً تمرکز ما روی این مسئله هست که همه مهره‌ها دارای ویژگی موقعیت (مثلاً `position`) و رفتار حرکت کردن به نام `move()` هستند. اگر بازیکن‌ها (که خودشان یک شئ دیگر هستند) به طور مستقیم به مقدار position دسترسی داشته باشند، نمی‌توانیم هیچ محدودیتی روی نوع حرکت مهره‌ها داشته باشیم.

مثلاً کاربر موقعیت مهره خود را از x به y تغییر می‌دهد. شاید این حرکت طبق قوانین بازی ما مجاز نباشد. پس نباید به بازیکن‌ها اجازه دسترسی مستقیم به موقعیت مهره را بدهیم. در این حالت کاربر متد `move()` را صدا زده و موقعیت مورد نظرش را اعلام می‌کند، اگر موقعیت خانه مورد نظر طبق قوانین بود، مهره حرکت می‌کند.

> کپسوله‌سازی برای ویژگی‌ها، متدها و حتی کلاس‌ها قابل تعریف است.

ارث‌بری کلاس‌ها در OOP

به کمک **ارث‌بری** (**Inheritance**) می‌توانیم کلاس‌ها را گسترش داده و ویژگی‌ها و رفتارهای جدیدتری به آن اضافه کنیم. در ۲ مثال زیر یک نمونه خیلی ساده از ارث‌بری در دنیای واقعی را به تصویر کشیدیم:

- ☑ همه ماشین‌ها نوعی «خودرو» هستند. همگی در و پنجره داشته و حرکت می‌کنند. اما ماشین‌ها به دسته‌های کوچک‌تری هم تقسیم می‌شوند. مثلاً از نظر سوخت مصرفی می‌توانند متفاوت باشند، که در این صورت عملکرد برخی قسمت‌ها متفاوت خواهد شد. دسته‌بندی دیگر بر اساس امکانات بسیار متفاوت (نظیر سقف بازشو یا دنده اتومات) است.
- ☑ همه جانوران نوعی «حیوان» هستند. آن‌ها راه می‌روند و غذا می‌خورند. در این بین، اسب‌ها با چهار پا راه می‌روند، کرم با خزیدن حرکت می‌کند و ماهی از باله‌های خود کمک می‌گیرند. غذای هر کدام از این‌ها هم متفاوت است.

اگر دقت کنید، همه ماشین‌ها ویژگی رنگ و چرخ دارند. اما خودروهای گازی، یک کپسول گاز و سیستم گازسوز اضافه بر سایر خودروها دارند. همین قضیه برای مثال حیوانات هم برقرار است.

در **برنامه نویسی شئ گرا** وقتی ویژگی‌ها و متدهای زیادی بین دو (یا چند) کلاس مشترک هستند، معمولاً یک کلاس بزرگ‌تر حاوی موارد مشترک در نظر گرفته و کلاس‌ها زیرمجموعه را از آن ارث‌بری می‌کنند.

> مشابه دنیای واقعی که هر فرزند تقریباً همه ویژگی‌های والدین خود را به ارث می‌برد، در برنامه‌نویسی نیز به همین شکل است. برخی از ویژگی‌های ارث گرفته در فرزند ممکن است تقویت یا تضعیف شده باشد و همچنین یک فرزند خصوصیات غیر مشترکی هم با والدین خود دارد. این قضیه در ارث‌بری برنامه‌نویسی شئ‌گرا نیز برقرار است.

ارث بری در برنامه نویسی

مثال برنامه شئ گرا مدیریت دانشگاه را در نظر بگیرید. در این برنامه، استاد، دانشجو و کارمند همگی از کلاس انسان هستند ولی تفاوت‌هایی در ویژگی‌ها و رفتارهای خود دارند؛ مثلاً:

- ☑ استاد می‌تواند نمره درس را تعریف کند ولی دو نوع دیگر نمی‌توانند.
- ☑ دانشجو می‌تواند درس را اخذ کند ولی این کار برای دو نوع دیگر معنایی ندارد.
- ☑ یک کارمند چیزی به نام ساعت حضور و غیاب دارد که دانشجویان ندارند.
- ☑ دانشجو دارای شماره دانشجویی است ولی استاد و کارمند دارای کد پرسنلی هستند که ممکن است ساختارش متفاوت باشد.

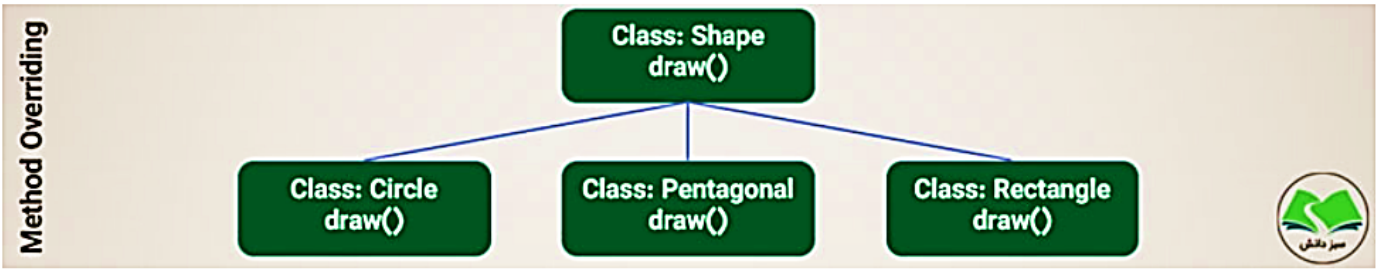
در این حالت در **برنامه نویسی شیء گرا**، ما یک کلاس کلی از نوع انسان ایجاد می‌کنیم که دارای متدها و خصیصه‌های مشترک است؛ یعنی راه می‌رود، حرف می‌زند و دارای نام، کد ملی و تاریخ تولد است. سپس کلاس‌های استاد و دانشجو از این کلاس ارث‌بری کرده تا توانایی‌های مخصوصشان پیاده‌سازی شود.

> بسته به نیاز سیستم، احتمالاً استاد یک نوع کارمند در دانشگاه است.برای اینکه **مفهوم ارث‌بری در برنامه‌نویسی** را بهتر متوجه شوید، سعی کنید در زندگی روزمره خود مثال‌هایی برای آن پیدا کرده و آن‌ها را تعمیم دهید.

مفهوم چند ریختی

چندریختی (**Polymorphism**) : بخوانید پُلی مورفیسم) به ما اجازه می‌دهد که متد دارای اجراهای مختلفی باشد. **دو نوع چند ریختی در برنامه نویسی شئ گرا داریم**. که در این بخش به طور مختصر با آن‌ها آشنا می‌شویم.

چند ریختی با Method Overriding : فرض کنید کلاسی برای شکل‌های هندسی داریم که یک متد رسم (`draw()`) دارد. کلاس‌های مستطیل، پنج ضلعی و دایره از این کلاس ارث‌بری کرده و چون نحوه رسمشان متفاوت است، این متد را **بازنویسی** می‌کنند.



بازنویسی متد در برنامه نویسی شیءگرا

چند ریختی با Method Overloading : در این حالت در برنامه نویسی شئ گرا از ارث‌بری خبری نیست. همچنین نام متد و نوع خروجی آن (امضا) تغییری نمی‌کند. صرفاً چند متد مشابه با آرگومان‌ها (پارامترهای ورودی) متفاوت در اختیار داریم.

خلاصه مفاهیم برنامه نویسی شی گرا

بحث برنامه نویسی شئ‌گرا **خیلی گسترده** و گاهی اوقات گیج‌کننده است! اما چون مدل‌سازی آن به دنیای واقعی شبیه است، درک مفاهیم شئ گرایی ساده می‌شود.

در این آموزش با زبانی ساده با مبانی برنامه نویسی شئ گرایی آشنا شدیم. مفهوم کلاس، شئ، ویژگی و متد را فهمیده و با ۴ اصل اساسی آن را بررسی کردیم.

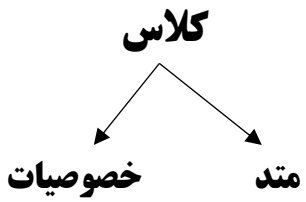
اگر می‌خواهید در مورد تاریخچه شکل‌گیری زبان‌های **برنامه نویسی شئ گرا** بخوانید، در ویکی‌پدیا اطلاعات مختصر و مفیدی ارائه داده است.

Object oriented

[شئ گرایی به بیانی دیگر]

در ادامه " 4 اصل شئ گرایی " را به بیانی دیگر بیان میکنیم.

معمولا برای داده ها در یک کلاس ، یک متد یا تابع تعریف میشود و فقط از آن متد میتوان به آن دسترسی پیدا کرد و دسترسی به آن برای همه آزاد نخواهد بود.



هر برنامه ای نیاز به تحلیل و طراحی ندارد چون ممکن است آنقدر ساده باشد که فقط نیاز به کدنویسی ساده داشته باشد.

نکته: فقط سیستم های مختلف یک نرم افزار که پیچیدگی دارند نیاز به تحلیل و طراحی دارند.

پیچیدگی نرم افزار (Complexity)

پیچیدگی یک سیستم و نرم افزار به 2 عامل بستگی دارد.

پیچیدگی نرم افزارها ، به افزایش تعداد و تنوع اجزا و رابطه ها بستگی دارند.

یادآوری: سیستم مجموعه ای از اجزا و رابطه ها است.

نرم افزارهای پیچیده ، تعداد اجزای سیستم زیاد و تنوع رابطه های بسیاری دارند. ممکن است که سیستم فقط اجزا و تعداد زیادی داشته باشد یا فقط تنوع و رابطه های بسیاری داشته باشد ، که در این صورت باز هم یک سیستم پیچیده است.

در سیستمی که پیچیدگی افزایش پیدا کند (با توجه به تکنیک ها) ، اثر آن در کاهش نظم مشاهده میشود ، و برای کاهش پیچیگی باید دسته بندی انجام دهیم.

هر چه پیچیدگی (باتوجه به اجزا و تکنیک ها) افزایش پیدا کند ----> نظم کاهش پیدا میکند ----> دسته بندی برای کاهش پیچیدگی انجام میشه

نگاه شئ گرایی سبب میشود ما نگاهی به دسته بندی در سیستم خود داشته باشیم.

Object Oriented یک پارادایم است / یعنی همه جا قابل استفاده است.

پارادایم شئ گرایی همه جا میتواند شکل گیرد که در آن اجزا مستقل از هم هستند.

پس درواقع وقتی میگفتیم سیستم نرم افزاری ، مجموعه ای از اجزا و رابطه ها است ، ماهیت آن اجزا به پارادایم بستگی دارد.

اصول شئ گرایی به طور کلی / 4 اصل شئ گرایی

4 اصل شئ گرایی را در صفحات قبل توضیح داده ایم ولی الان آن ها را به بیانی دیگر (بیانی که سر کلاس گفته شده) بیان میکنیم.

1) Abstraction (نگاه انتزاع / نگاه کلی):

درمورد یک موضوع کلی صحبت میکند ، بدون اینکه به جزئیات اشاره کند. (جزییات را نادیده میگیریم.)

خصوصیات نگاه انتزاعی:

1- نگاه ما را به موضوع ساده تر میکند.

2- توجه به بخش های مهم تر.

3- نگاه کلی و ساده به موضوع برای قابل فهم شدن آن.

4- نادیده گرفتن جزئیات.

2) Encapsulation (کپسوله سازی):

مخفی سازی یا پنهان سازی / بسته بندی / محافظت

با کپسوله کردن اطلاعات ، دسترسی به آنها را محدود میکنیم و جلوی بسیاری از خطاها را میگیریم.

اجزا را میتوان به عنوان یک کپسول در نظر بگیریم و اجزا در کپسول ها بسته بندی میشوند و رابطه های تعریف شده برقرار ارتباط بین اجزا را فراهم میکنند. **هر کلاسی باید بتواند اجزاء و رابطه های داخلی کلاس (شناسه و رفتارها) را از دید کلاس های دیگر مخفی کند. !!!!**

این اصول به عنوان یک قالب الزامی نیستند و ما به عنوان یک ایده در تحلیل و طراحی از آنها استفاده میکنیم.

مثلا در یک فضا ما میتوانیم یک متغیر را کپسوله کنیم و یک متغیر دیگر را سراسری داشته باشیم و پرایوت نکنیم.

3) Modularity (پیمانه بندی / واحد بندی / ماژول بندی):

در ماژول بندی ما میتوانیم هر بخشی را به صورت جداگانه انجام دهیم.

در پیمانه بندی باید جوری عمل کنیم که ماژول ها یک همبستگی داخلی داشته باشند و این همبستگی همواره بیشتر شود و ماژول ها ارتباط بیرونی کمتری داشته باشند.

باید وابستگی خارجی ماژول ها به هم کمتر باشد و وابستگی درونی آنها به هم بیشتر باشد.

سوال مهم کنکور ارشد ← ماژول ها
- بیشترین همبستگی داخلی (پیوستگی) / Cohesion
- کمترین وابستگی بین پیمانه ها و اجزا / Coupling

باید در داخل بسته ها به هم پیوستگی و انسجام داشته باشند ولی سایر پیمانه ها نیاز و وابستگی کمتری به هم داشته باشند. / رابطه ی بین 2 ماژول با فراخوانی متد فراهم میشود.

4) Hierarchy (سلسله مراتب):

یکی از راه های ساده کردن کار ایجاد سلسله مراتب است و تعداد دسته بندی ها کم کم به ترتیب بیشتر شوند.

از اجزای تشکیل دهنده میتوان به سلسله مراتب رسید.

سلسله مراتب:
- Aggregation به معنای بخشی از / (is a part of)
- Inheritance (وراثت) به معنای نوعی از / (is a kind of)

حالا با توجه به اینکه مطالب بیشتری یاد گرفتیم ، پیچیدگی نرم افزاری را از دید گسترده تری میبینیم.

پیچیدگی نرم افزاری:
- تعداد ماژول زیاد
- تنوع ماژول زیاد
- تعداد زیاد رابطه
- تنوع زیاد رابطه
- افزایش سلسله مراتب

سوال: بر اساس چه قانونی ، سیستم را به ماژول ها بشکنیم ؟؟

جواب: مدیریت ساختار

Information System

اساسی ترین نقش Information System به عنوان یک جز:

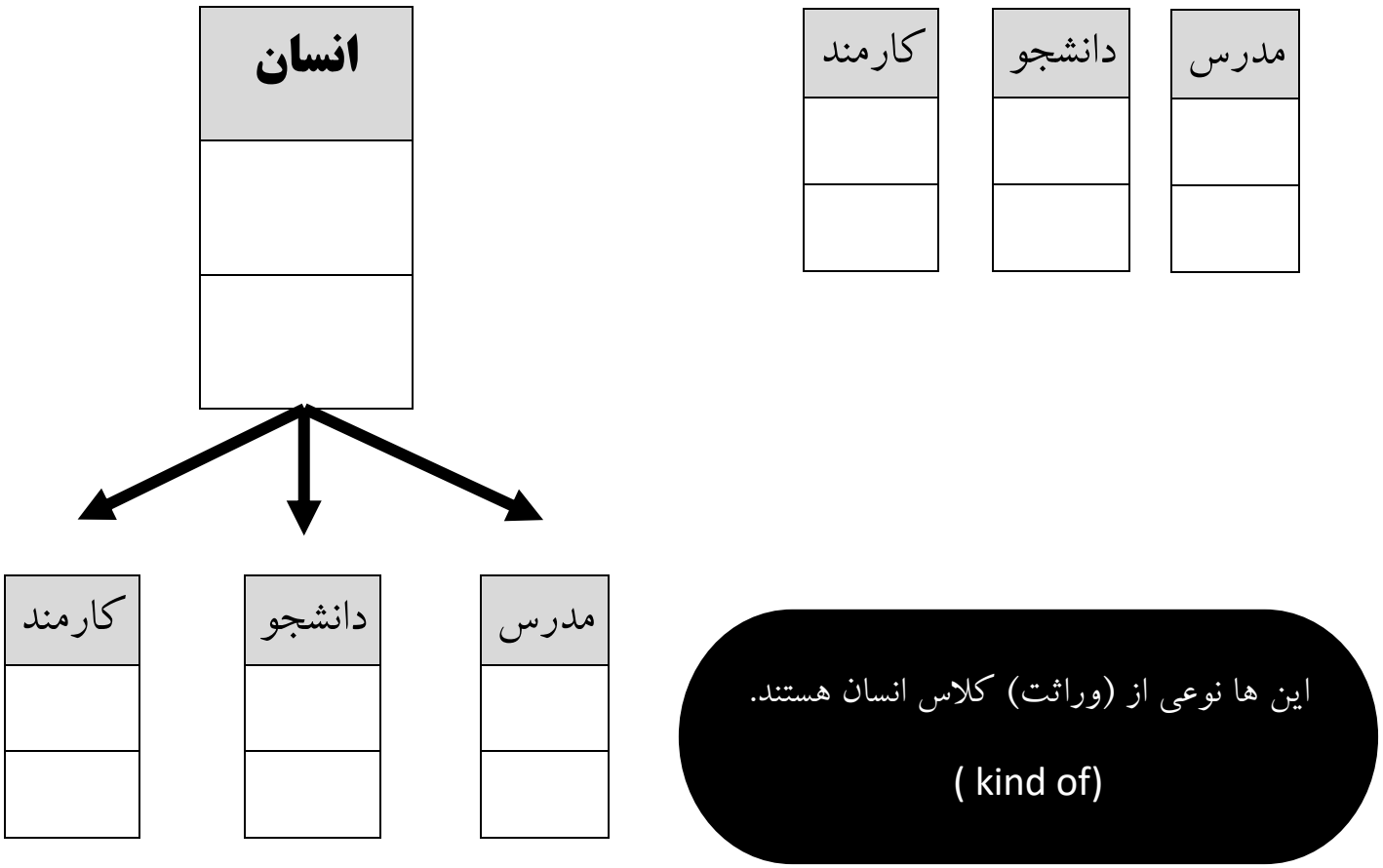
1) خدماتی

2) تولیدی / مونتاژ

3) بازگردانی

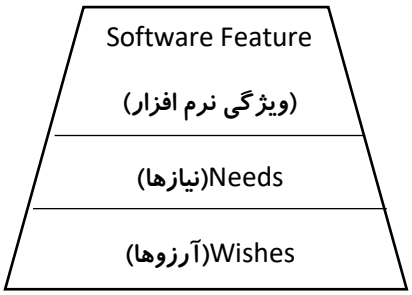
4) پشتیبانی

نکته: کلاس انسان ، میتواند وراثت های مختلفی را داشته باشد که عبارت اند از:



Requirement Gathering / جمع آوری نیازمندی ها :

جمع بندی نیازمندی های سیستم موجود + نیاز ها و الزامات سیستم نرم افزاری جدید



software requirements

نیاز های نرم افزاری

کلیه ی قابلیت هایی است که نرم افزار باید داشته باشد.

نیازمندی های نرم افزار در مهندسی نرم افزار توصیف ویژگی ها و ویژگی های سیستم هدف است. نیازمندی ها انتظارات کاربران از محصول نرم افزاری را منتقل می کند. الزامات می تواند آشکار یا پنهان ، شناخته یا ناشناخته ، مورد انتظار یا غیر منتظره از دید مشتری باشد.

software requirementها به 2 دسته تقسیم میشوند :

1) دسته ی اول که به آنها نیازهای **Functional** (نیازهای عملیاتی) میگویند.

این نیازها با یک سری فعالیت های محدود ، قابل پیاده سازی و اجرا میباشند. (use case model)

2) و دسته ی دوم که به آنها نیازهای **non-Functional** (نیازهای غیر عملیاتی) میگویند.

این نیازها :

1- به سادگی قابل پیاده سازی نیستند.

2- به تکنولوژی ، نرم افزار و زبان برنامه نویسی مورد استفاده وابسته است.

چند دسته از non-Functional :

کارایی (Performance)

میزان در دسترس بودن

میزان جا به جا پذیری

بخشی از امنیت (Security)

Business Modeling / مدل کسب و کار

یک مدل کسب و کار برای سیستم های بزرگ و معمولا تجاری است که در آن یک شرکت به تولید درآمد یا سودی از عملیات شرکت میپردازد. به زبان ساده ، روشی است که شرکت ها برای تولید پول از آن استفاده میکنند.

(برای سیستم های بزرگ و تجاری (ممکن است تجاری هم نباشد.))

مراحل Business Modeling به شرح زیر است:

1) مقدمه و چکیده

که سیستم چه کاری انجام میده + معرفی کامل و دقیق سیستم و بخش های مختلف و کاربردهای آن

2) اهداف

هر سازمانی یک سری اهداف کلی با نام Vision دارد ، این خدمات ممکن است به زیر مجموعه ها تقسیم شود. Vision : چشم اندازی است که میخواهیم به آن برسیم. یعنی هدفی است که میخواهیم برایش تلاش کنیم. (مثلا Vision مدرسه ، تعلیم و تربیت است.)

3) نقش های سازمانی / (Role) / (با امکان ارث بری)

4) واحد های فیزیکی و واحد های سازمانی

5) فرآیندهای سازمانی (Business Process)

مجموعه ای مرتب از فعالیت های سازمانی است. ما در سیستم کارهایی که باید انجام شوند را شناسایی میکنیم. مثلا در سیستم کتابخانه ، امانت دادن یک کتاب ، یک فرآیند سازمانی خواهد بود.

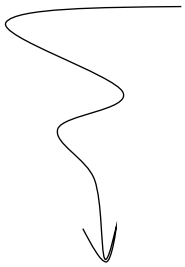
فرآیندهای سازمانی (Business Process) **، شامل مراحلی است که عبارت اند از:**

- کد فرآیند
- نام فرآیند
- شرح مختصر (در حد 4 تا 6 خط است.)
- event
- سناریوی فرآیند
- Resource ها / منابع
- from
  - کد فرم
  - نام فرم
  - شکل
  - فیلدهای روی فرم
  - بررسی های روی فرم

---

6) Glossary یا Data Dictionary

لغت	مفهوم لغت در سازمان مربوطه



توضیح "دیتا دیکشنری" در صفحه ی بعد است. !!

Glossary یا Data Dictionary

دیتا دیکشنری ، مانند یک لغت نامه برای واژه ها و عبارات مورد استفاده در یک سیستم است.

یک روش اصلی برای تحلیل جریان داده ها و همین طور مخازن داده ها در سیستم است.

که اجزا و ساختار داده ها را تعریف میکند و برای ارجاء چیستی و ماهیت داده ها و اطلاعات ، به آن مراجعه میکنیم. و در حقیقت کار آن جمع آوری ، هماهنگ سازی ، تغییر و یکسان سازی داده های متنوع برای ایجاد یک مفهوم واحد در سازمان است.

سوال: دیتا دیکشنری در کجا استفاده میشود ؟؟

جواب: در همه ی قسمت های سیستم به خصوص مراحل تحلیل و طراحی.

دلایل استفاده از Data Dictionary :

1) مستند سازی

2) حذف دوباره کاری ها و تکرارها

3) راهنمایی منابع اطلاعاتی

4) کنترل توصیف داده ها

5) بهبود بخشیدن به کنترل و دانش درباره ی منابع

6) ارائه ی تعاریف دقیق از داده ها

7) تامین طرح های امنیتی / 8) اعتبا ر بخشیدن و تسهیل و اصلاح کردن

برای بیان سناریوی فرآیند ، میتوان از روشی به نام Swimming Line استفاده میکنند.

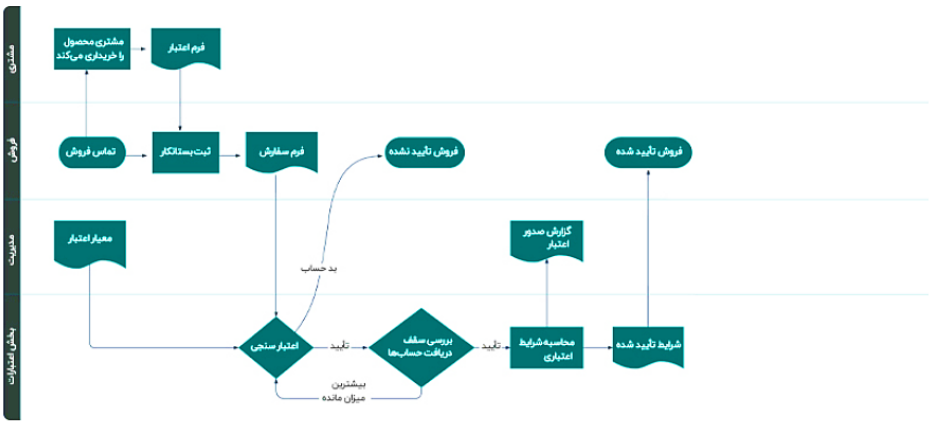
Swimming Line / خط شنا

برای الان میخواهیم ،Swimming Line را برای شما توضیح دهیم.

این خط شنا ، در زمینه این‌که چه کسی، چه کاری را در فرایند نمودارهای گردشی انجام می‌دهد، شفافیت بیشتری ایجاد می‌کنند.

نمودار swimlane چیست؟

نمودار swimlane نوعی نمودار گردشی است که وظایف هرکسی را در فرایند، مشخص می‌کند. این نمودار با استفاده از تشبیه خطوط استخر، با قراردادن مراحل یک فرایند، میان خطوط افقی یا عمودی خطوط شنای یک کارفرما، گروه کاری، یا یک بخش، باعث شفافیت در کار و مسئولیت‌پذیری می‌شود. این تشبیه نمایانگر روابط، ارتباطات و تبادلات بین این خطوط است و می‌تواند عوامل زائد، تکراری و ناکارآمد را مشخص کند.



این نوع از نمودار به نام‌های «نمودار راملر و بریش» (Rummler-Brache diagram) یا «نمودار میان‌کارکردی» نیز مشهور است؛ گاهی به نمودار swimlane، نوارهای کاربردی نیز می‌گویند. این خطوط، عناصر ارزشمندی هستند که مدل و نشانه‌گذاری فرایند کسب‌وکار (BPMN و BPMN ۲.۰) یا طرح نرم‌افزار همتای آن، یعنی زبان مدل‌سازی یکپارچه (UML) و نمودارهای جریان فرایند (PFDs)، از آن‌ها استفاده می‌کنند.

نمادهای استانداردی که در تمامی این نمودارها استفاده شده است، همراه با خطوط شنا، نمایشی دیداری از مسئولیت‌ها را در طول فرایند ارائه می‌کنند که به‌آسانی خوانده می‌شوند.

اهداف و فواید نمودار swimlane

در سازمان‌های امروزی که داری چندین گروه کاری یا بخش هستند، در بسیاری از موارد، ساخت نمودار می‌تواند کمک‌کننده باشد:

این نمودار، امکانی را فراهم می‌کند که کارمندان بخش‌های مختلف، از کار یکدیگر آگاه باشند. نمودار swimlane و خطوط شنایی که در نمودارهای دیگر استفاده شده‌اند، مشخص می‌کنند که کدام مرحله از فرایند یا کدام زیرفرایند، باید به کدام عامل واگذار شود.

با توضیح این مسئله در نمودار، امکان شناسایی زوائد و ناکاآمدی‌های دیگر، در خطوط موارد تکراری و تنگناها فراهم می‌شود؛ این کار معمولا مراحل غیرضروری و تکراری را آشکار می‌کند؛ مثلا هنگامی که بخش‌های مخلتف در حال انجام کار مشابهی هستند.

این نمودار، تأخیرهای موجود در فرایند و محدودیت‌های ظرفیتی را درون خط شنای خاصی مشخص می‌کند تا بتوان به آن‌ها مراجعه و آن‌ها را برطرف کرد. این روش، عملکرد و کیفیت کار را بالا می‌برد و هزینه و کارهای غیرضروری را کاهش می‌دهد. شما می‌توانید از نمودار swimlane دومی، برای بهترین روش مدل‌سازی در راستای ساخت فرایند یا توضیح بروز تغییر در شرایط، مانند تغییر کارمندان یا تکنولوژی استفاده کنید.

نمودار swimlane نیز مانند نمودارهای دیگر، در مقایسه با روایت توصیفی، اطلاعات را بهتر انتقال می دهد. نمودار swimlane ، مانند روشی برای یکپارچه‌سازی فرایندها، بین تیم‌ها و بخش‌های مختلف استفاده می‌شود که در طول زمان، منجر به فرایندهای کامل‌تری می‌شود.

ویژگی‌ها و عناصر نمودار swimlane

استفاده از هریک از نمادهای نمودار گردشی ایجادشده، خطوط افقی، عمودی یا موازی، خطوط شنای مراحل فرایند را بر اساس عواملی مانند بخش گروه کاری یا حتی سیستم اطلاعاتی، گروه‌بندی می‌کنند. هر خط با همان عامل طبقه‌بندی می‌شود. گاهی اوقات، نه همیشه، خطوط افقی کاربردی‌تر هستند، زیرا عرض صفحه دسک‌تاپ کامپیوترها، از ارتفاع آن‌ها بیشتر است.

مراحل فرایند در خطوط مخصوص به خود، توضیح داده می‌شوند و ارتباط بین خطوط نیز نشان داده می‌شود. این مورد چگونگی ارتباط متقابل عوامل مختلف را نشان می‌دهد که باعث حفظ کارایی فرایند می‌شود.

کسب‌وکار ۲.۰ (BPMN ۲.۰ diagram)

در مدل‌ها و نمادهای فرایند کسب‌وکار ۲.۰ (Business Process Model and Notation ۲.۰) یا به‌اختصار BPMN ۲.۰، استخر یا خطوط شنا یکی از ۴ نوع عنصر مورداستفاده است. استخر نمایانگر شرکای اصلی در فرایند است. این امکان وجود دارد که استخری متفاوت یا بخش مختلف دیگری در شرکت وجود داشته باشد، اما هنوز به فرایند مرتبط باشد.

خطوط شنای داخل استخر، فعالیت‌ها و جریان‌هایی را برای نقشی خاص یا شرکت‌کننده مشخصی نشان می‌دهد و مشخص می‌کند که چه کسی مسئول انجام چه قسمتی از فرایند است.

۳ نوع دیگر از این عناصر، عبات‌اند از:

- جریان اشیا: رخدادها، فعالیت‌ها و دروازه‌ها؛
- اشیا متصل‌کننده: جریان توالی، جریان پیغام و پیوند؛
- مصنوعات: شیء داده، گروه و حاشیه‌نویسی.

Activity Diagram / نمودار فعالیت

یکی از ابزارهای UML است. یک نوع پیشرفته و حرفه ای فلوچارت است. (نوع پیشرفته فلوچارت)

ابزاری برای مدل کردن سناریوی فارسی و برای نمایش مجموعه ای از عملیات است.

- ✓ فعالیت عملی است که به وسیله انسان یا کامپیوتر انجام می‌شود و در دیدگاه پیاده سازی، متد است.
- ✓ نمودار فعالیت یک فلوچارت است که برای نمایش جریان کنترل از یک فعالیت به فعالیت دیگر به کار می رود.
- ✓ از بسیاری جهات شبیه state diagram است.
- ✓ نمودار فعالیت برای نمایش جریان کار و نمایش رفتاری که پردازش‌های موازی دارند مناسب است.
- ✓ در این نمودار صرف نظر از اینکه فاعل رفتار چه کسی است، میتوان رفتار را به خوبی با تقدم و تاخّر و بیان شرط های لازم نمایش داد.

در صفحه ی بعد ، اکتیویتی دیاگرام را به طور کاملللل بررسی میکنیم !!!!

اکتیویتی دیاگرام Actvity Diagram چیست؟

همانطور که در مقاله یوزکیس دیاگرام Usecase Diagram دیدید، یوزکیس ها به شما میگویند که نرم افزار قرار است چه کاری قرار انجام دهد ولی اکتیویتی دیاگرام Activity Diagram به شما خواهند گفت که این کارهای باید چگونه انجام شوند تا نیازمندی های نرم افزار برآورده شود. در نمودار اکتیویتی قرار است بصورت سطح بالا مشخص شود که چه فعالیتهایی باید انجام شود تا هر کدام از این کارها و usecase ها انجام شوند.در این دیاگرام مراحل و step ها و زنجیره انجام کارهای مشخص خواهند شد.

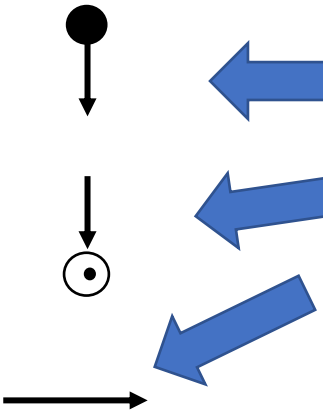
میتوانیم اکتیویتی دیاگرام را به عنوان مدلسازی بیزینس و لاجیک Business Logic Modeling دانست. بیزینس مدل زنجیره ای فعالیت ها و Task ها است که با هم منجر به یک کاری خواهند شد. بعضی از نرم افزارهای مدلسازی پراسس های تجاری Business Proccess Management Tools یا به اختصار BPM ها ابزاری را برای شما فراهم میسازند که فرآیند تجاری هر بخشی از کسب و کار خود را مدلسازی و ترسیم کنید.

اکتیویتی دیاگرام تنها نموداری است که در معماری نرم افزار در بخش Process Architecture ترسیم خواهد شد. قواعد و Syntax هایی که در اکتیویتی دیاگرام استفاده می شوند بسیار ساده و مانند فلوچارت هستند و notation های خیلی قابل فهمی برای عمومی دارند و به همین دلیل این نمودار بهترین دیاگرام برای ارائه به ذینفعان سطح بالای نرم افزار مانند هست مدیره و Board Member ها هستند. در اکتیویتی دیاگرام شما قرار است مراحلی که برای انجام هر یوزکیس لازم است را مدلسازی کنید.

شروع هر نمودار اکتیویتی یک دایره توپر هست که بهش Initial Node گفته میشه و سپس با فلش هایی روبه پایین حرکت میکنیم که این فلش ها رو Edge یا Path میگیم و یه سری باکس های مستطیلی که بهشون Action گفته میشه هم برای نشون دادن کاری هست که در هر قدم باید انجام بشه. علامت لوزی یعنی تصمیم گیری و Desecion که نشون میده که این فلو بعد از این نقطه دو حالت براش ممکنه پیش بیاد. مثلا یا کاربر لاگین هست و یا کاربر لاگین نیست و یه لوزی دیگه هم داریم که فلش ها بهش وارد میشن، یعنی چند تا حالت مختلف به یه لوزی وارد میشن و این یعنی از این مرحله به بعد فلوهاشون یکی میشه ولی در اولی فلش ها از لوزی خارج میشدند. به حالت دومی Merge میگیم و در نهایت Final Node رو دایم که یه دایره توپر هست که دورش یه دایره گرد داره.

به فلش های ورودی به یک نود Incoming Edge و به خروجی Outcoming Edge گفته میشه.Action ها قدم های فعال در یک Process هستند و هر اکشنی میتونه یه چیز محاسباتی یا منطقی باشه، مثلا محاسبه مالیات یا بررسی لاگین بودن و ... ضمنا میشه دور هر نمودار Activity یک باکس هم کشید که بهش میگن Activity Frame و اختیاری هست و معمولا نمیکشند و اگر در یک صفحه قرار شد بیش از یک نمودا اکتیویتی باشه اینکار میکنیم و داخل این اکتیویتی فریم هم اسم اکتیویتی رو Activity Name بصورت بولد مینویسیم.

در Outgoing Edge های هر Decssion روی edge ها باید داخل براکت شرطی که اتفاق افتاده رو بنویسیم وبه اینها Guard Condition میگیم و البته این خروی ها بصورت true- false هستند و فقط یکی از دو حالت ممکن را دارند. دقت کنید اگر بیش از دو تا guard condition داشته باشیم باید خیلی دقت کنیم که فقط یکی از همه اونها درست باشه

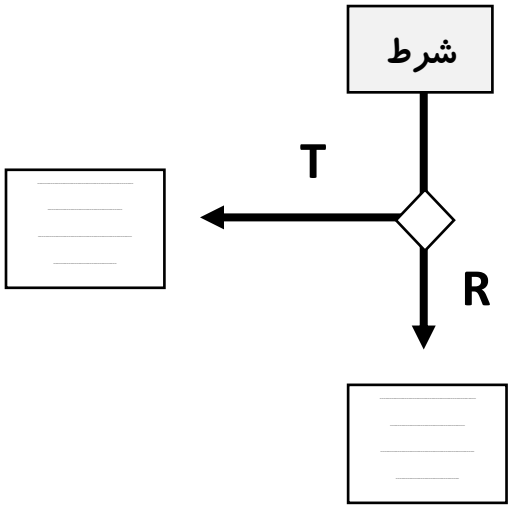


در اکتیویتی دیاگرام : برای **نقطه شروع** از شکل مقابل استفاده میکنیم.

برای **نقطه پایان** از شکل مقابل استفاده میکنیم.

برای **کنترل جریان** از شکل مقابل استفاده میکنیم.

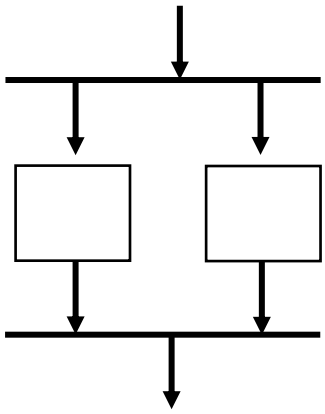
در اکتیویتی دیاگرام ، یک قسمت به نام شرط داریم که حالت های مختلفی برای آن پیش می آید. **که در صفحه ی قبل آن را توضیح دادیم و الان فقط شکل آن را میبینیم :**



سناریوی فرآیند

- شروع کننده
- Role ها
- Swim Line
- متن
- Model
  - با حداکثر جزییات
    - کلیه خطاها به صورت کامل بررسی میشوند.
    - (کلیه ی بررسی ها به صورت کامل است.)
- با چه Resource هایی (منابع) و چه عملیاتی روی آنها رخ میدهد.
- انجام موازی کارها (با روش چنگال) / fork - join

گاهی هم کارها میتوانند به صورت موازی انجام شوند. (انجام موازی کارها) / Fork- join



---

2 نکته ی خیلیییی مهم

1) یوزکیس دیاگرام به ما میگوید که نرم افزار قرار است چه کاری انجام دهد.

2) ولی اکتیویتی دیاگرام به ما نشان میدهد که این کارها را چگونه انجام دهیم.

Domain Modeling / مدل سازی دامنه

درواقع دامنه همان چیزی است که میخواهیم روی آن کار کنیم ، تا مشکلاتی که توی کار داریم را رفع کنیم و قوانین کار ، فرآیندها و سیستم های موجود را بررسی میکنیم.

- به ما میگوید که چه چیزی هست و چه چیزی نیست !!

- و به ما میگوید که نحوه ی ارتباط درون و بیرون چگونه است !!

و میتوان هم اینجوری گفت که مجموعه ای انتزاعی از دانش ، برای حل مشکلی است که با آن سر و کار داریم.

مدل دامنه، سیستمی انتزاعی است، که جنبه‌های منتخب، تأثیرگذاری و فعالیت در حوزه دانش را توصیف می‌کند و از این مدل می‌توان برای حل مشکلات مربوط به دامنه، استفاده نمود. مدل دامنه نمایانگر مفاهیم معنی‌دار از دامنه، در دنیای واقعی می‌باشد، که باید در نرم‌افزار مدل شوند. در زبان مدل‌سازی یکپارچه، از نمودار کلاس برای نشان دادن مدل دامنه استفاده می‌شود.

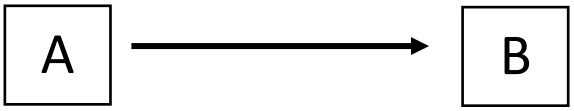
ما سیستم ها را از 2 دیدگاه میتوانیم بررسی کنیم. (بررسی سیستم از 2 دیدگاه) که آن دو دیدگاه عبارت اند از :

1) Dynamic (پویایی)

2) Static (استاتیک)

تعریف "رابطه وابستگی" :

بین زیرسیستم ها ،یک وابستگی ای وجود دارد که باعث میشود هر تغییری در یک زیرسیستم ، در زیرسیستم دیگر هم تاثیر و تغییر داشته باشد.



هر تغییر در B باعث تغییر در A میشود.

نکته: سعی میکنیم ارث بری را بکار نبریم. / اگر لازم شد خیلی کم و محدود استفاده میکنیم.

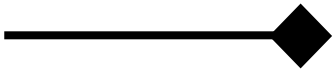
- تحلیل کلاس های سیستم را انجام نمیدهیم. (چه چیزی با چه چیزی رابزه دارد.)

- رفتار برای کلاس ها در نظر نمیگیریم.

- از شناسه ها فقط به موارد اصلی اشاره میکنیم. (3 الی 5 شناسه)

- رابطه Association (همکاری / اتحاد)

- رابطه Aggregation (اجتماع / تجمع) 

- رابطه Composition (ترکیب بندی) 

ارث بری: یعنی ویژگی ها و رفتارهای یک کلاس از کلاس بالاتر به ارث گرفته شود. (یک کلاس از یک کلاس دیگر متولد شود.)

دقیقا مانند تولید مثل انسان ، که فرزندان ویژگی ها و رفتارهایی را از والدین خود به ارث میبرد. البته هر فرزند میتواند ویژگی ها و رفتارهای منحصر به فرد خودش را نیز داشته باشد.

Actor: در دنیای نرم افزاری هر چیزی خارج از سیستم نرم افزاری که با سیستم در تعامل است و درخواست انجام کاری را دارد و سیستم در مقابل آن درخواست ، کاری انجام میدهد.

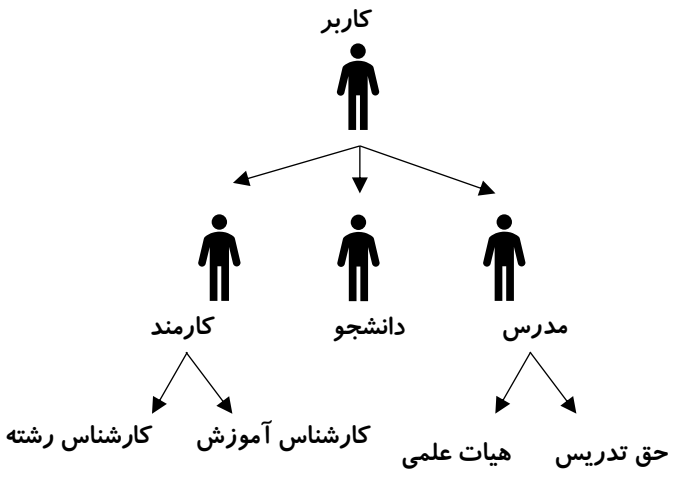
انواع اکتورها عبارت اند از:

1) افراد – اشخاص

2) سیستم نرم افزاری دیگر / بخشی دیگر از نرم افزار خودمان

3) زمان (مثلا پشتیبان گری یا ریکاوری و سیو خودکار اطلاعات)

نقش های سازمانی و فرایندهای سازمانی شبیه ارث بری یک درخت است:



در دنیای نرم افزار:

Use Case ها : مجموعه ای مرتب و پایان پذیر از فعالیت های نرم افزار به صورت یکپارچه (یک تکه) را یوزکیس میگویند. / همه ی انواع گزارش ها یوزکیس هستند.

هر UC (Use Case) باید یکپارچه اکتور داشته باشد. / هر یوزکیس باید یا چند اکتور داشت باشد.

منابع استخراج UC ها:

1) فرآیندهای سازمانی (فعالیت های انجام شده در یک سازمان)
2) رفتار Actor ها
3) کلاس های Domain Modeling
4) گزارش های نرم افزاری
5) فرم های نرم افزاری

یاد آوری

فرآیندهای سازمانی: یک سری از کارها و رفتارهایی که در طول پروژه انجام میدهیم تا ما را مرحله به مرحله به هدف پروژه نزدیکتر کند که در دنیای موجود / سیستم فعلی است.

Use Case: کارهایی هستند که ما باید مرحله به مرحله بین کاربر و سیستم برای رسیدن به هدف انجام بدیم. (در دنیای نرم افزاری است. / و بخشی از فرآیند سازمانی است.)

مجموعه ای مرتب و پایان پذیر از فعالیت های نرم افزار به صورت یکپارچه (یک تکه) را یوزکیس میگویند.

Use Case Diagram: نموداری است که شامل خود یوزکیس و اکتور و ارتباط است.

تفاوت فرآینده های Use Case ها :

1) **اولین تفاوت که مهم ترین تفاوت هم است ، آن است که "فرآیندهای سازمانی" در دنیای موجود و سیستم فعلی هستند ، / ولی" یوزکیس ها "در دنیای نرم افزاری هستند.**

2) **تفاوت بعدی این است که "فرآیندهای سازمانی" به صورت یکپارچه نیستند ولی "یوزکیس ها" به صورت یکپارچه هستند.**

3) **تفاوت بعدی این است که "یوزکیس ها" بخشی از یک فرآیند سازمانی هستند ولی فرآیندهای سازمانی بخشی از یک "یوزکیس" نیستند.**

Use Case از دیدگاه پیچیدگی :

از دیدگاه پیچیدگی پیاده سازی ، Use Case ها را میتوان به 2 دسته اصلی و غیر اصلی تقسیم کرد.

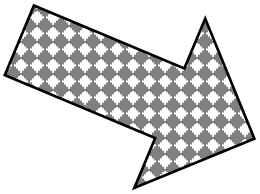
از دیدگاه پیچیدگی پیاده سازی

1) Use Case **های اصلی ---------> حدود** 10 **درصد اند.**

2) Use Case **های غیر اصلی**

UC های اصلی حدود 10٪ از یوزکیس ها هستند. ما برخی کارها را برای یوزکیس های اصلی انجام میدهیم که برای یوزکیس های غیر اصلی انجام نمیدهیم.

UC های غیر اصلی انواع مختلفی دارند که عبارت اند از:

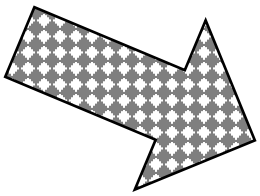


1) گزارشات

2) CRUD ها

3) کلاس های Domain

پیچیدگی UCها به عوامل مختلفی بستگی دارد که عبارت اند از:



1) تعداد خط های کد مورد نیاز

2) پیچیدگی الگوریتم

3) تازگی و نو بودن UC ها

4) پیچیدگی محاسباتی

5) پیچیدگی فرم

6) Swim Line , Business سنگینی داشته باشد.

نکته: تفاوت UC عادی با UC پیچیده ، در نحوه ی مستند گیری است.

برای هر یوزکیس ، باید یک سری مستندات و مواردی را رعایت کنیم که الان به آن ها اشاره میکنیم.

مستندات مورد نیاز برای UC ها (به طور کلی) :

1) نام UC

2) کد UC

3) توضیحات (Note)

4) فرم ها (form)

5) لیست اکتورها (Actor)

6) با کدام کلاس Domain Model چه کاری انجام میدهد.

مستند سازی به ازاری هر UC اصلی:

1) نام UC

2) کد UC

3) توضیحات (Note)

4) شرط ها

Precondition (پیش شرط) : شرط هایی که قبل از انجام یوزکیس باید صحیح شده باشند.

لیستی به صورت فارسی است. / مثلا پیش شرط های امانت کتاب...

Postcondition (پس شرط ها)

5) فرم ها (Form)

6) سناریوی UC