جلسه اول:

در XP از کارت CRC برای تشخیص کلاس ها استفاده میشود که فرض میشود داریم از شی گرا استفاده میکنیم. UP فرآیند یکپارچه هست. اجایل بدرد همه حوزه ها نمیخورد. اگر سطح بحرانیت بالا باشد مثل سیستم های پزشکی که جان انسان در خطر هست، که به اصطلاح life critical هستند و چابک مناسب نیست کلا در scale بزرگ مناسب نیست. SCRUM یک چیز basic هست. golden hammer یعنی یک چیز خیلی روتین را که ساده هست مرتب استفاده کنیم واسه هر استفاده ای در صورتی که scrum خودش قبول کرده خیلی جا ها مشکل دارد و مناسب نیست. پس باید آلترناتیو های مختلف را بشناسیم. UP دقت بالایی دارد و بسیار بزرگ هست و شخصی سازی میکنیم قسمت هاییش را برای پروژه خودمان و مدل سازی UML در آن توصیه میشود و برای پروژه های سنگین مناسب هست و مدل سازی اجباری هست. UP ساده شده RUP بود که انحصاری واسه اون شرکت بود، UP واسه پروژه های سنگین هست که واسه پروژه تعداد برنامه نویس بالا هست. اول از agile شروع میکنند نشد از UP استفاده میکنند. هم refactor و باز آرایی کد و الگو های طراحی را بخوان خودت. Gangof4 را بخوان. دقت کن فاز های UP وقتی اجرا شوند در نهایت یک release آماده و بدست کاربر میرسد. UP تکراری افزایشی و ۵ تا workflow یا discipline گفته میشود و در هر فاز یک سری تکرار داریم و در هر iteration این ۵ تا اجرا میشوند و حالت ترتیبی دارند. دو تا آخری در هم تنیده هستند. در UP ۴ فاز داریم inception elaboration construction, transient، اولی آغاز و دومی یعنی وارد جزئیات و چهارمی یعنی انتقال واسه اولی ریسک را میسنجیم منابع مورد نیاز و ببینیم آیا ارزش دارد پروژه آغاز شود یا نه و امکان پذیر هست یا نه در فاز دوم دنبال تحلیل هستیم و بدانیم چیستی سیستم چیست بدون توجه به پیاده سازی کلا میخواهیم بدانیم دقیقا چی هست و بعد برای ساخت نقشه بریزیم. تحلیل بر تقدم پا قدم دارد. نیازمندی ها را تا ۸۰ درصد بدست میاوریم. انتزاعی هستند و هیچ جزئیاتی در آنها نیست. در همین فاز دوم یک مقدار وارد طراحی میشویم و یک راه حل یا solution ارائه میدهیم، و یک نمونه نرم افزاری با حداقل نیازمندی ها میسازیم و تصمیمات تکنولوژی مثل دیتا بیس چه زیر ساخت چه میان ابزاری تعیین میشود و UI شکل گرفته است دقت کن فقط نیازمندی های پایه هست و در مرحله سوم USE CASE ها اعمال میشوند روی همین سیستم و نیازمندی های واقعی اجرا میشوند. این نمونه اولیه prototype نام دارد ولی دور ریختنی نیست. این پرو تایپ را میسازیم که تکامل بدهیم و معماری ما را دارا هست مثل اینکه از چه تکنولوژی و دیتا بیسی استفاده میکنیم چه سیستم عاملی چه محیطی، یک معماری سطح بالا هم داریم که کامپوننت های سیستم نهایی را مشخص میکند. تحلیل دو مرحله دارد طراحی هم دو مرحله دارد طراحی پایه که سرسری میگوییم سیستم چطوری هست و طراحی تفضیلی که کامل گفته میشود کلاس ها و کامپوننت ها و سایر موارد. در فاز دوم تحلیل تفضیلی و طراحی معماری (همون مقدار پایه یعنی وارد راه حل شدیم.) در فاز آغاز تحلیل مقدماتی انجام دادیم. در مرحله سوم کاملا وارد راه حل میشویم و طراحی تفضیلی انجام میدهیم و اینکه دقیقا از چه نیازمندی هایی و چه سیستم هایی قرار هست اضافه کنیم کاملا مشخص میشود کار های مقدماتی در فاز دوم انجام شده است اینکه از چه سیستمی استفاده کنیم چه سرویس هایی و چه محیطی و در فاز سوم فقط iteration میزنیم و نیازمندی های بالا تر را اعمال میکنیم. در انتهای فاز سوم سیستم آماده برای ارائه به کاربر داریم در فاز چهارم نرم افزار ساخته شده را از محیط ایجاد به محیط کاربر منتقل میکنیم واسه همین میگوییم انتقال بعد سعی میکنیم به ثبات برسانیم. در جریان کاری تحلیل یا قلمرو مسئله را مشخص میکنیم و اینها جز ذات سیستم ها هستند و با هر پیاده سازی اینها توشون هستند که کلاس های تحلیل یا کلاس های قلمرو مسئله نام دارند. Object یک نمونه از کلاس هست و روابط در کلاس تعریف میشود روابط بین کلاس ها مثلا و این روابط به اشیا دو طرف هم منتقل میشود. کلا یعنی بین دو اشیا رابطه بوده

بعد این را در کلاس رابطه ایجاد کردیم ماهیت این رابطه چیست در ادامه. اشیا در زمان اجرا و در عمل کار میکنند کلاس ها در سطح بالا هستند و کلی تر هستند. اشیا موجودیت فعال هستند نه کلاس ها.

بعضی محدودیت ها در سطح کلاس تعریف میشود بعضی ها هم نمیشود در جریان کاری تحلیل و کلاس های تحلیل و روابط بین آنها در آن جلسه بررسی میکنیم. با پکیج دیاگرام میگوییم برای قسمت بندی کلاس دیاگرام وقتی که کلاس دیاگرام سنگین هست. از پکیج دیاگرام برای مدل سازی سیستم در تحلیل، و میگوییم این سیستم ما چگونه هست و هر زیر سیستم یک پکیج هست و use case ها از تعاملات بین اشیا ساخته میشوند در نمودار توالی هست و اشیا در زمان اجرا چگونه با هم ارتباط برقرار کنند تا یوز کیس محقق شود یعنی کلاس های تحلیل و اشیا اینها باید بتوانند با هم ارتباط برقرار کنند. اگر نه یعنی ناقص هست. در دیاگرام توالی نشان میدهد که ابجکت ها چگونه با هم درگیر هستند و پیام میفرستند. ارسال پیغام هم که یک فعالیت هست در اکتیویتی دیاگرام مشخص میشود. پکیج دیاگرام فقط در تحلیل استفاده میشود و در طراحی تبدیل میشود به کامپونت دیاگرام در UP اتفاق میفتد. در جریان کاری طراحی میرویم سراغ کلاس های طراحی یعنی این کلاس هایی که جز قلمرو مسئله هستند باید در راه حل هم نمود پیدا کنند. پکیج دیاگرام فقط در قلمرو مسئله یعنی فاز تحلیل هست و هر کلاسی که در قلمرو مسئله باشند در هر طراحی و پیاده سازی ظهور پیدا میکنند. قلمرو جواب یک سری چیز ها داره که اصلا جز قلمرو مسئله نیست و ما باید بهش اضافه کنیم مثل UI یا مسائل مربوط به پایگاه داده، کلاس های مربوط به ارتباط با میان افزار و کلاس های مربوط به ماشین. این کلاس ها مال راه حل های طراحی هستند و مال تحلیل نیستند و کلاس دیاگرام طراحی خیلی بزرگتر از تحلیل هست چون همه جزئیات هست. روابط در طراحی برگرفته شده از روابط در تحلیل هست ولی جزئیات بیشتر هست. refinement یعنی جزئی گویی و اضافه کردن جزئیات و برعکس abstraction هست که کلی گویی هست. وقتی میگویند چیزی لاجیکال هست مال قلمرو مسئله هست ولی وقتی میگویند فیزیکال مال قلمرو جواب و پیاده سازی هست، کامپونت یک جز نرم افزاری هست و ارتباط آن با سایر کامپونت ها از طریق interface ارتباط دارد داخل آن معلوم نیست و قابل جا به جایی هست، کامپونت ها به هیچ چیزی در فاز تحلیل متناظر نیست چون خاص قلمرو جواب هست مثل UI. دقت کن هر چیزی که در قلمرو مسئله باشد نگاشت پیدا میکند در قلمرو جواب ولی جواب یک سری چیز بیشتری دارد. GRASP میگوید چگونه عملیات ها را اختصاص بدهی به کلاس ها که ۹ تا الگو هست. طراحی بر اساس قرارداد برای افزایش قابلیت اطمینان هست و مبتنی بر تفکر شی گرا هست. GoF را بخوان حتما. سایت RSPA.COM پرس من هست و اطلاعات تخصصی میگیریم.

نکات اولیه از همین جلسه:

UP فرآیند یکپارچه هست دقت کن امروزه جز غیر چابک ها و شی گرا ها امروزه فقط از این استفاده میکنند ولی چابک ها در همه جا استفاده میشود. برای سیستم های در مقیاس بزرگ یا سیستم های life critical اصولا نمیتوان از چابک ها استفاده کرد چابک برای پیچیدگی و بحرانیت بالا مناسب نیست. SCRUM هم توی چابک ها خیلی ابتدایی هست. golden hammer ها مثل همین SCRUM یک سری anti pattern هستند که یک سری روش های رایج استفاده میشود ولی نتیجه منفی هست. اول چابک اگر نشد استفاده کنیم میایم سمت شی گرا ها. UP همان RUP هست که یک نسخه سبک وزن تر و رایگان هست.

جلسه دوم:

De facto یعنی در عمل استاندارد هست یعنی بدون استانداردی و قانونی رایج شده است ولی de jure یعنی قانونی استاندارد شده است. UML هر چیزی که در یک محصول نرم افزاری هست حتی اگر یک نرم افزاری باشد که میان راه تولید یک نرم افزار دیگر تولید

بشود را امکان نشان دادن دارد، DFD و دیتا فلو دیاگرام برای نشان دادن انتقال داده نمایش داده میشود و هر نود یک پردازنده هست و کمان های انتقال داده هستند و هیچ کنترلی روی آن نیست. این نمودار تقدم و تاخر روی آن نیست. به این نمودار میگویند فانکشنال یعنی نمودار هایی که تقدم و تاخر در آنها مشخص نیست و فقط میگویند چه کارهایی انجام میشود. مثلا در یک اداره کارمند ها پردازنده یا پراسس هستند، و اسنادی که رد و بدل میشود همان جریان داده هست. روش های ساخت یافته یا structured که این نمودار باعث شکل گیری آنها بود البته امروزه شی گرا جای آنها را گرفته است. یکی از معروف ترین آن SA/SD بود analysis design بود. برای DFD هیچ نمودار جایگزینی برای آن وجود ندارد مثلا میتوانی با اکتیویتی دیاگرام نشان بدهی ولی فانکشنال نیست و اندازه DFD قوی نیست. حوزه DFD را function oriented گفته میشد فانکشن ها در قالب پراسس ها نشان داده میشدند و جریان هم با DFD بوده قبل اینکه شی گرا بیاد، یعنی چه کارهایی با چه انتقال داده ای بدون تقدم و تاخر نشان میدادیم و برای اون زمانی که زبان های procedure بودند و دید فانکشنال داشتیم به سیستم، یک زمانی خیلی خوب بود چون از روی DFD ما به ساختار برنامه پراسیجال میرسیدیم. دو عملیات روی DFD انجام میشد transformو transaction آنالیز که الگوریتم بودند و خروجی یک نمودار ساختار بود و ساختار برنامه را نشان میداد یک درخت بود که بعد تبدیل میشد به یک دایمند یا لوزی چون از پایین نود ها جمع میشد به بالا. اکتیویتی نزدیک ترین جایگزین DFD هست. در اکتیویتی دیاگرام تقدم و تاخر داریم که بهش object oriented flow chart هم گفته میشود و با یک کمان نشان میدهیم این کار که انجام شد بعدش کدام کار انجام میشود و اکشن ها بعلاوه ترتیب آن نشان داده میشود به اینها که تقدم و تاخر را هم نشان میدهند رفتاری یا behavioral گفته میشود در مقابل فانکشنال مادلینگ. یک وجه سوم هم داریم که به نام ساختاری هست که بحث کار ها که انجام میشود نیست بلکه میگوییم یک کل از چه اجزایی تشکیل شده است و اینها چه روابطی با هم دارند. کلاس دیاگرام ساختاری هست ERD ساختاری هست در پایگاه داده. بعضی نمودار های UML بهترین از نوع خودشون نیستند بعضی ها شی گرا هستند بعضی ها نیستند که در اینها میتوان شی گرایی را تزریق کرد. به صورت کلا business modeling دو بخش دارد: Business domain modeling میگوید که چه عناصر اطلاعاتی در سیستم وجود دارد و ارتباط اینها چیست شبیه ERD که میگوید موجودیت های داده ای در سیستم چی هستند و ارتباط آنها با هم چیست. نمودار ساخته شده ساختاری هست یا از کلاس دیاگرام یا از ERD استفاده کنیم که دومی در UML جایی ندارد. یا میتوانی از یک notation خاص استفاده کنیم مال همین business modeling که اون هم مال OMG هست یک گروهی از شرکت های بزرگ نرم افزاری هست که یک سری قواعد تعریف میکنند و استاندارد میشود در نرم افزار ها و به صورت de facto میشود. OMG یک زبان مدل سازی خاص به نام BPN تعریف کرده است، که همان notation ها هست و خیلی کامل هست و خیلی بهتر از UML هست. BPN خاص بیزنس مادلینگ هست ولی باز UML استفاده میشود. BPN فقط مثلا برای اتوماسیون ها استفاده میشود که خیلی دقیق بخواهند کاری کنند چون UML در این حوزه خیلی دقیق نیست. در business domain modeling عناصر اطلاعاتی تشکیل دهنده و روابط بین آنها را تعریف میکنید مثلا در دانشگاه دانشجو و استاد و درس وجود دارد تا سیستم بتواند کار بکند پس باید مدل سازی بکنیم توسط ERD اما در business process modeling که توسط DFD که تقدم و تاخر را نشان نمیدهد میتوانیم برای مدل سازی استفاده کنیم و در این قسمت ما فرآیند کسب و کار را مدل سازی میکنیم یعنی چرخش کار داخل سیستم چطوری هست و همچنین میتوانیم با اکتیویتی دیاگرام استفاده کنیم که تقدم و تاخر را نشان میدهد. DFD تک بعدی هست و حواس را به تقدم و تاخر پرت نمیکند کاملا فانکشنال هست کار ها را نشان میدهد به همراه انتقال ارتباط بین آنها، و هر فرآیند به صورت موازی و مستقل از دیگران اجرا میشوند و به صورت اکتیو در حال اجرا و مبادله داده دارند. DFD از اکتیویتی دیاگرام بهتر هست چون تمرکز بیشتری دارد و چند وجه را بهم قاطی نکرده است.

وقتی بخواهی رفتاری و ساختاری و ... را با هم ترکیب کنی پیچیده تر میشود. DFD به UML اضافه نشود با اینکه بهترین هست چون با شی گرایی و دید شی گرا تضاد دارد. EUP ۴ فاز دارد و ۱۷ تا discipline دارد برخلاف اینکه در up کلا ۵ تا داریم. و برای مدیریت سازمانی و مسائل مربوط به آنها هست. در EUP میگوید برای BPN از DFD استفاده کن بعد اطلاعات را استخراج کن و برای مدل سازی یک سازمان DFD را پیشنهاد میکنند چون دقیقا شبیه به سازمان هست چون کارمند ها پراسس ها و کمان ها انتقال داده بین آنها هست. UML2 رایج است. روش CRC برای استخراج کلاس ها استفاده میشود که برای هر کلاس مجموعه مسئولیت هاش را در میاری و برای هر مسئولیت میگویی با چه کلاس هایی باید همکاری کند تا بتواند مسئولیت هاش را انجام دهد پس مجموعه همکاران هم بدست میاد و درون کارت هایی ثبت میکنند. اجایل ها مدل گریز هستند ولی به CRC توصیه میکنند. Formal specification روش های مبتنی برای بیان دقیق سیستم ها و برای صحت سنجی یا verification استفاده میشوند برای عملکرد نرم افزار مخصوصا برای سیستم های بحرانی. سیستم های e یا essential critical مال سیستم هایی هستند که اگر خراب شود یا دچار خطا شود ضرر هنگفت مالی به طور ورشکستگی شرکت اتفاق میفتد گفته میشود. Life critical ها را l میگویند که جان انسان در خطر هست در آن سیستم ها و discretionary ها که پول روزانه سازمان در خطر میفتد یعنی دچار خطا شود سازمان باید خسارت بدهد ولی این ضرر زیاد نیست به اینها سیستم d میگویند. یک سری دیگر comfort critical هستند یعنی راحتی کاربر خدشه میبیند و نا خوشایند هستند و به اینها سیستم های c میگویند برای e و l چون خطا قابل تحمل نیست باید از formal specification استفاده کنیم زمان و هزینه بسیار بالایی میبرد و کار هر کسی نیست. UML میاد از ERD برای طراحی دیتابیس استفاده میکند ERD نسخه قدیمی تری از کلاس دیاگرام هست که در آن تمام موجودیت ها داده ای هستند و در آن عملیات وجود ندارد و attribute دارند. تصمیم گرفته شد تا همه متدولوژی ها از UML برای مدل سازی ها استفاده کنند. UP برای فرآیند های توسعه نرم افزار هست. UP چون تصویب نشد همیشه رقیب داشته ولی UML هیچ رقیبی ندارد. مدل یک نوع نمود از انتزاع یا abstraction هست. OPD را نگذاشتند وارد UML شود چون خیلی پیچیده بود. UML از یک سری دیگر متدولوژی دیگر تشکیل شده است که نمودار های آنها مورد استفاده قرار گرفته اند. و نسل اولی ها شی گرا را وارد عرصه کردند. متدلوژی های سومی مثل RUP که سنگین تر هستند و از ترکیب نسل دوم و اول هستند که ادغام شده اند. fusion نسل دومی خیلی مهم هست. catalysis هم یک متدولوژی نسل سوم و کامپوننت بیس و سنگین هست و تاثیر گذار روی UML بود. OPEN هم نسل سومی هست و سنگین بود و البته الان این ۲ ۳ تا که گفتم کنار رفته اند و استفاده نمیشود جز این ۲ بقیه در عکس نسل اول و دومی هستند. در RUP هم از UML استفاده شده است. و بعدا catalysis و OPEN هم اومدند از UML استفاده کردند و همه اینها غیر متداول هست و کلا برای هدف آموزش هست و در صنعت استفاده ای ندارد امروزه از RUP, UP, EUP استفاده میشود. امروزه از ورژن UML2 میاد بیرون . UML به ۲ دسته تقسیم شده است نمودار های رفتاری و ساختار که در عکس مشاهده هست. در ساختاری ها هر کدام با دید خودش میگوید سیستم از چه عناصری تشکیل شده است و ارتباط بین عناصر هست چیست. دقت کن use case diagram چون میگوید چه فانکشن هایی یا چه وظیفه مندی هایی دارد ولی تقدم و تاخر آنها و ترتیب آنها گفته نمیشود یک نمودار فانکشنال هستند ولی جز رفتاری ها گذاشته اند. interaction دیاگرام ها خودشان یک نوع دیاگرام هستند که برای تعامل پیغام بین ابجکت ها دیده میشود. اکتیویتی دیاگرام: برای نشان دادن رفتار پراسیجال و موازی و رویه ای استفاده میشود. کلاس دیاگرام: برای نشان دادن کلاس ها و ویژگی هاشون (attribute, operation) ها هستند و روابط بین آنها را نشان میدهد communication: یک نمودار تعاملی هست که تاکید آن روی ساختار بین ابجکت ها هست ولی رفتار را هم نشان

میدهد. کامپوننت: سیستمی و نرم افزار های کامپوننت را نشان میدهد. composite structure دیاگرام: برای نشان دادن کلاس های مرکب و پیچیده که داخل ابجکت های آنها ابجکت وجود دارد یعنی ابجکت های تو در تو وجود دارد یا nested استفاده میشود.

Deployment: نشان میدهد فایل ها یا اجزای نرم افزاری در حال اجرا (artifact) چگونه روی پلتفرم میشینند یعنی بستر معماری آن چیست و چگونه کامپوننت های نرم افزاری در قالب artifact، یا فایل های زمان اجرا چگونه میشینند روی بستر و معماری سخت افزاری و نرم افزاری آن چیست. Interaction overview: یک سطح بالاتری از ترتیب را نشان میدهد بین رفتار های درشتانه معمولا ترتیب اجرای یوزکیس ها را نشان میدهد چون داخل یوزکیس دیاگرام اینها را نشان نمیدهیم البته ترتیب اجرای مجاز. Object و پکیج دیاگرام: نمودار های ساختاری هستند قبلا ذیل نمودار های دیگه بودند ولی الان مستقل هستند. sequence دیاگرام: برای نشان دادن پیغام رسانی و تاکید روی ترتیب و رفتار و توالی انتقال پیام هست. state: دیاگرام یعنی یک ابجکت یا بخش های درشتانه سیستم چگونه حالت آن عوض میشود در نتیجه اومدن وقفه، timing: که از سخت افزار آمده است محدودیت های زمانی را برای انجام کار ها نشان میدهد و این محدودیت باید پیروی شود و از سیستم های الکترونیکی اومده است داخل نرم افزار برای نشان دادن اینکه چگونه حالت های مختلف ابجکتی و تبادل پیام ابجکت ها انجام میشود چه محدودیت های زمانی باید رعایت شود و برای سیستم های که response time توی آنها محدودیت دارد مثل real time یا event driven مهم هست. Use case نشان میدهد اکتور های بیرونی چه انتظاری از سیستم ما انتظار دارند و چه وظیفه مندی را از سیستم انتظار دارند. ابجکت دیاگرام میگوید که این ابجکت ها در زمان اجرا چگونه به همدیگر دید دارند. یعنی در یک لحظه خاص یک اسنپ شات از وضعیت ابجکت ها را نشان میدهیم. چون کلاس دیاگرام به صورت گویا در وضعیت اجرا مشخص نمیکند ابجکت ها چگونه هستند. مثلا وضعیت آفساید در فوتبال یک نمونه هست که نمیتوان با کلاس دیاگرام نشان داد. اگر یک ابجکت با خودش یا یک کلاس با خودش رابطه داشته باشد یک رابطه انعکاسی هست. نمودار پکیج واسه پارتیشن بندی کلاس دیاگرام و معماری سیستم مشخص میشود. در up از کلاس دیاگرام بزرگ به پکیج دیاگرام نمیسازند معمولا top-down ساخته میشود یعنی یک سیستم را متشکل از یک سری زیر سیستم میبینند و اونجوری پکیج دیاگرام میسازند یعنی واحد های مختلف مثلا آموزش یا سلف و ... ، UI و DB مال قلمرو جواب هستند یعنی در طراحی و در تحلیل جایی ندارند. به چیستی سیستم کار داریم به چگونگی پیاده سازی کاری نداریم. Composite هم که گفتیم ابجکت های تو در تو داریم. کامپوننت دیاگرام نرم افزاری خالص هست و کامپوننت ها به درون همدیگر دید ندارند و encapsulation عالی میدهد و انعطاف پذیری عالی میدهد و تغییر دادن سیستم خیلی آسان هست.

جلسه سوم:

کامپوننت ها از طریق interface ها با هم در ارتباط هستند و از درون همدیگر خبری ندارند و دقت کن اگر یک کامپوننت خراب شد با همان interface ها میتوان یک کامپوننت دیگر جایگذاری کرد کامپوننت های داخلی هم میتوانیم داشته باشیم یعنی دو کامپوننت درون یک کامپوننت باشند. Message queue واسه وقتی هست که سر سرور خیلی شلوغ باشد و میاد مسیج ها را صف بندی میکند. Artifact یعنی عنصر نرم افزاری در زمان اجرا، پکیج دیاگرام معماری و ساختار سطح بالا را نشان میدهد و نشان میدهد در قلمرو مسئله در فاز تحلیل چه زیر سیستم هایی داریم. در طراحی این پکیج ها به یک یا چند کامپوننت مپ میشود و یک سری کامپوننت داریم که هیچ چیزی نگاشت نمیشوند و جدا هستند مثل ui یا database. Deployment: این دیاگرام نمود کامپوننت های نرم افزاری هست. در آن یا اجزای سخت افزاری هستند مثل web server یا اجزای نرم افزاری نشسته روی آن سخت افزار. به دیوایس های نرم

افزاری execution environment گفته میشود. ارتباط بین مسیر مبادلات داده و لینک های بین سخت افزار و پروتکول های آن نشان داده میشود در این دیاگرام در واقع معماری سخت افزاری هست و پایین ترین سطح این سیستم هست یک سطح بالاتر میان افزار ها هستند که بستر نرم افزاری ما را ساختند و ما بستر نرم افزاری را از EE نشان میدهیم. این کامپوننت های سخت افزاری دقیقا مپ نمیشوند به یک عنصر در زمان اجرا بلکه صرفا به یک تعدادی فایل و کامپوننت تبدیل میشوند که روی این بستر نرم افزاری نشسته میشوند اصطلاحا گفته میشود که کامپوننت های ما مپ میشوند به یک تعدادی artifact، که اصطلاح خاص UML هست برای بیان اینکه داریم یک سری عنصر زمان اجرا را نشان میدهیم. و هر کدام از کامپوننت های این دیاگرام مپ میشوند به یک یا چند artifact. این تناظر یک به چند هست سمت کامپوننت ۱ و artifact چند یعنی به بیش از ۱ artifact مپ میشوند مثلا میتواند یک فایل ارشیو یا اجرایی باشد و به صورت توزیع شده در نود های مختلف قرار میگیرند. مثلا herculesclient.exe یکی از همین artifact ها هست یکی از مهم ترین دیاگرام ها هست چون در قلمرو راه حل نشان میدهد بستر سخت افزاری و معماری آن چگونه هست و بستر میان افزاری مثل سیستم عامل چطوری نشسته است و روی آن کامپوننت های سیستم ما نشسته اند و در قالب artifact ها بروز پیدا میکنند، و در هر جاهای مختلف قرار میگیرند یک artifact در جاهای مختلف قرار میگیرد. این ۳ نمودار جنبه معماری دارند که اولین آن پکیج هست، که ساختار سطح بالا و درشتانه را نشان میدهد در متدولوژی UP صرفا واسه تحلیل هست و نشان میدهد در قلمرو مسئله یک سیستم از چه زیر سیستم هایی تشکیل شده است، زیر سامانه های مختلف در فضای قلمرو مسئله و صرفا در تحلیل در طراحی این پکیج ها نمود آن میشود کامپوننت ها و هر پکیج به یک یا چند کامپوننت در فضای طراحی مپ میشوند بعضی ها هم به هیچ چیزی مپ نمیشوند چون خاص قلمرو راه حل یا جواب هستند مثل ارتباط با DB یا UI یا ارتباط با ماشین و آنها قطعا خودشان را در طراحی نشان میدهند و تحلیل نشان نمیدهند و در کامپوننت دیاگرام یک سری کامپوننت داریم که به هیچ چیزی در فضای تحلیل مپ نمیشود چون مال قلمرو جواب یا راه حل هست این نمودار معماری است منتها در فضای طراحی یا حل مسئله و کاملا تمرکزش روی نرم افزار ما هست. نمودار پکیج یا بسته، در تحلیل نشان میدهد سیستم ما شامل چه زیر سیستم هایی هست و کامپوننت در فضای طراحی یا حل مسئله نشان میدهد سیستم ما از چه کامپوننت هایی درست شده است و نمودار deployment هم مال معماری هست منتها صرفا به نرم افزار محدود نیست اول بستر سخت افزاری بعد بستر نرم افزاری بعد نمود های کامپوننت های ما یا artifact های ما روی آن قرار میگیرند و صرفا به نرم افزار هدف معطوف نیست و بستر زیرین را نشان میدهد. در این دیاگرام ۲ نوع داریم ۱. سطح کلاسی که رابطه بین کلاس ها را نشان میدهد ۲. سطح ابجکتی که رابطه بین instance ها را نشان میدهد. ۱. در سطح کلاسی هر کدام از انواع دیوایس ها وee ها و artifact ها فقط یک بار دیده میشوند چون نوع کلی مورد توجه قرار میگیرد. در ابجکتی اگر از یک artifact چند بار وجود دارد و روی نود های مختلفی قرار دارد، اگر از یک دیوایس چند تا وجود دارد، چند بار ذکر میکنی و ارتباط هر کدام با هم را میگی کاملا عناصر مختلف artifact بر روی نود های مختلف قرار میگیرد نشان میدهی. ابجکت دیاگرام وقتی هست که کلاس دیاگرام کافی نیست مثلا روابط ابجکت ها در زمان اجرا. به دلیل همین خصوصیت که یک سطح کلاسی داریم که به طور کلی مشخص میشود و سطح ابجکتی که چون هر کدام ابجکت های مختلف دارد هر کلاس در هر لحظه خاص با استفاده از این نمودار در زمان اجرا گفته میشود همچنین چگونه با هم ارتباط دارند. Deployment دیاگرام به خاطر همین خصوصیت کلاسی و ابجکتی، در کلاسی نوع کلی هست ولی در ابجکتی چون هر کدام instance مختلف در زمان اجرا دارد و میخواهیم در یک زمان خاص چگونه با هم ارتباط دارند را با ابجکت دیاگرام گفته میشود. اونایی که خواستی تو زمان اجرا بگی برو با ابجکتی آن. دقت کن نموداری که کارها را نشان میدهد ولی ترتیب آنها را نشان نمیدهد به این نمیگویند رفتاری به این میگویند فانکشنال اما UML برای سادگی به یوزکیس دیاگرام گفته است رفتاری. این نمودار خاص نشان میدهد مشتریان بیرون سیستم کیا هستند به این ها actor گفته میشود و از سیستم انتظار سرویس دارند یوز کیس ها همان سرویس ها هستند.

با خطوط بین اکتور و یوز کیس نیاز آن اکتور نشان داده است. نشان میدهد اکتور ها از چه سیستم چی میخواهند مثلا یک اکتور خاص در یک سرویس خاص به عنوان فعال کننده درگیر باشد یعنی خودش از سیستم انتظار دارد یا اکتور ثانویه باشد یعنی وسط اجرای یوزکیس از آن بخواهد اطلاعی بدهد. یا بعد از اتمام یوز کیس به او اطلاعی داده میشود همه اینها با یک خط نشان داده میشود. به این communication link گفته میشود. بقیه انواع روابط میتوانند در یوزکیس ها باشند یا نباشد مثلا یک رابطه include هست وقتی دو یوزکیس در یک مرحله داخلی با هم مشترک هستند این مراحل مشترک استخراج کشیده میشود در قالب یک یوزکیس جدا تعریف میشود و کارهایی مشترکی در دو یوزکیس داریم. و یک جوری duplication را از بین میبریم و به صورت صریح بیان کردیم این دو تا بخش مشترک دارند از طریق include به یوزکیس های پایه. رابطه extend یعنی یک یوزکیس پایه داریم که در شرایط استثنایی میتوان یک سری مراحل بهش اضافه شود. در این نمودار هیچ ویژگی نمیبینیم که ترتیب اجرای این یوزکیس ها چیست. یوزکیس ها را در یک جدول هم گفته میشود. و برای هر یوزکیس پیش شرط پس شرط primary actor ها کیا هستند (یعنی کیا اون یوزکیس را فعال میکنند) و secondary actor: یعنی کسایی هستند که در جریان یوزکیس درگیر میشوند و خودشون فعال کننده نیستند ضمنا مشخص میکنیم event flow چی هست یعنی جریان وقایع چی هست یعنی مراحل اجرای یوزکیس چی هست یعنی در جریان اجرا کدوم اکتور چه کاری به چه ترتیبی انجام میدهد و اثر آن روی سیستم چی هست. یا کدوم کار را سیستم و کدوم کار را اکتور انجام میدهد. منطق و مراحل یوزکیس در این جا آورده نمیشود بلکه در جدول ثبت میشود. همچنین لحاظ میشود که alternative flow events چی هست یعنی اینکه چه کار هایی باعث میشود که ما از جریان اصلی وقایع خارج شویم، مثلا وسط کار بزند کنسل. دقت کن main یکی هست منتها منطق اجرای alternative ها درون دیاگرام نماید و در کنار جدول میاد دقت کن جداول جدا دارند هر کدام و با جداول main جدا هست. رابطه دیگر رابطه genspeck هست که یک رابطه is-a هست و در ارث بری وقتی بدانیم فرزند مثل پدر نیست و رابطه ارث بری را اجرا میکنیم که این یک ارث بری هست صرفا. و این خطر خیلی عمده هست. در رابطه is-a زیر کلاس نوعی از ابر کلاس ها هست و instance زیر کلاس همان instance ابر کلاس هست اما در توارث این شکلی نیست. توارث میتواند بدون is-a باشد توارث در بطن is-a هست یعنی اون باشد توارث هم هست اما هر رابطه ای توارثی is-a نیست. مثل عنکبوت و اسب هر دو خوردن دارند پس میتوانی اشیا این را با اشیا آن یکی کنیم و عنکبوت را زیر کلاس اسب میکنیم و خوردن خوابیدن را عنکبوت هم دارد بعد لازم میشود یورتمه و چهار نعل را هم اضافه کنیم به عملیات یا به اسب ولی در عنکبوت موضوعیت ندارد پس این عملیات هایی که به ارث میرسد را خالی میکنند با overwrite کردن و refused bequest میکنند یعنی میراث مردود یعنی از توارث استفاده کردیم بدون اینکه رابطه is-a وجود داشته باشد. بعضی وقتا attribute ها هستند بعضی وقتا operation ها. ساختاری های توارثی بالاترین میزان coupling را دارند یعنی زیر کلاس به شدت به سوپر کلاس مربوط هست و اگر در سیستم شی گرا به این موارد دقت نکنیم باعث مشکلات میشود. Shotgun surgery: یعنی یک جای سیستم را میزنی درست کنی n جای دیگه خراب میشود. وقتی coupling در سیستم بالا باشد یک جا را تغییر بدهی n جای دیگر نیاز هست تغییر کند پس باید باز آرایی کنی. divergent change: تغییر واگرا یعنی یک کلاس به دلایل بی ربط عوض شود مثلا یک گرفتن پرینت عوض میشود این هم عوض میشود و ui عوض میشود این هم عوض میشود مشخص هست این کلاس یک وجهی نیست و چند وجهی هست ما میخواهیم کلاس های چند وجهی باشند. Maintenance: ۴ نوع دارد: corrected maintenance: تصحیحی یعنی وقتی یک باگ پیدا میشود و باید بر طرف کنیم. Perfected maintenance: یعنی وقتی که قابلیت اضافه میکنیم به سیستم. Adapted maintenance یعنی بستر عوض میشود و سیستم باید وفق پیدا کند با سیستم جدید. Prevent maintenance یعنی اجتنابی یعنی میدانیم در آینده یک

مشکل برای سیستم لحاظ میشود و از الان پیشگیری میکنیم. این واسه وقتی هست که سیستم داریم و نصب شده است حالا بحث نگهداری آن هست. پس باید دقت کنیم coupling بالا نرود تا بتوانیم انعطاف پذیری سیستم را نگه داریم. همیشه به maintainability و flexibility کد باید توجه کرد. UML به شدت تاکید دارد وقتی از وراثت استفاده کن که is-a وجود داشته باشد. یا میگوییم ساختار genspeck که داخلش is-a دارد برقرار هست و instance فرزند را به جای پدر میتوان استفاده کرد و کلاس ها زیر نوعی از کلاس پدر هستند. genspeck در نمودار یوزکیس وجود دارد و بین اکتور ها و بین یوزکیس ها میتواند نشان داده شود. در یوزکیس چه وظیفه مندی که سیستم باید داشته باشد را نشان میدهیم. Activity diagram: در شکل خالص شی گرا نیست چون اصلا شی توش نیست، شبیه فلوچارت هست یعنی میگوییم چه اکشن هایی انجام میشود و چه انتقال کنترلی داریم و اینکه مثلا اون افراد چیکار میکنند و مراحل مختلف و ترتیب اجرا مشخص میشود. ترتیب نشان داده میشود و کمان های بین نود ها برای انتقال کنترل هست خطوط پارتیشن actor ها هستند خودش کلش یک یوزکیس هست. برنچ و فورک و جوین هم داریم نمودار قوی برای نشان دادن هر منطق رفتاری هست و برای نشان دادن جریان یک یوزکیس طرفین درگیر چه کاری را انجام میدهند یعنی چه یوزکیسی هست اکتور ها کیا هستند خود سیستم چی هست و این سیستم چه بخش هایی از کار یوزکیس انجام میدهند همچنین اکتور ها چه کار هایی را انجام میدهند رفتار شرطی و منطق شرطی هم از طریق decision node ها نشان داده میشود عامل انجام کار هم مشخص میشود مثل اکتور مشتری این کار ها را کرده هست و هر کدام از اجزا مثل اکتور ها بخش خاص خود را دارند پارتیشن مخصوص به خود و اسم عامل انجام را هم میگذاریم و نمودار قوی میشود یعنی میگیم کدوم اکتور کدوم کار را انجام میدهد. main flow را هم میگوییم یعنی بعد از ساختار جدولی تبدیل به این دیاگرام میگوییم. اکتیویتی دیاگرام برای نشان دادن منطق هم به کار میرود مثلا ترتیب عملیات داخل یک کلاس یا ترتیب کار ها در یک عملیات کلا هر منطق اجرایی را میتواند نمایش دهد. مثلا برای سیستم آموزش اول در سیستم آموزش فعلی business process modeling میکنیم و جریان گردش کاری را بدست میاوریم قدیم با DFD بود الان با اکتیویتی دیاگرام هست و مدل سازی میکنیم توسط اکتیویتی دیاگرام میگیم چه کار هایی در سیستم انجام میشود عوامل کیا هستند و کار چگونه در سیستم میچرخد، کدوم واحد ها چه کاری را انجام میدهند و داخل واحد ها چه کارمند های چه کارهایی را انجام میدهند. بعد از روی این یوزکیس های سیستم را در میاوریم یعنی میگیم اگر سیستم ما به عنوان عامل وارد شد چه کارهایی بهش باید سپرده شود و اون کار ها میشود یوزکیس برای سیستم ما. دنیای واقع را میتواند مدل کرد. منطق داخل یک کلاس را میشود گفت. منطق داخل یک کلاس هم گفته میشود. همه کاربرد های اکتیویتی دیاگرام هستند. مثلا داخل سیستم ما در زمان اجرا چه ابجکت هایی وجود دارد، و داخل همین سیستم ما مثلا atm machine یک پارتیشن بهش میدهیم بعد اسم میدهیم بعد میگیم این ابجکت ها چگونه کار ها را رد میکنند و میگوییم یوزکیس چگونه توسط ابجکت های داخلی محقق میشود و به نمودار شی گرا تبدیل میشود چطوری با مشخص کردن ابجکت هایی که عامل انجام کار ها خواهند بود در زمان اجرا و ابجکت ها را تزریق میکند و اکشن ها میتوانند ابجکت تولید کنند و به همدیگر پاس بدهند که در object flow میبینیم این دیاگرام در زمان اجرا هست چه ابجکت هایی دارد، و یوزکیس چگونه توسط ابجکت هایی داخلی محقق میشود، ابجکت ها عامل انجام کار ها خواهند بود در زمان اجرا. گردش ابجکت ها بین اکشن ها را هم نشان دهیم و به همدیگر پاس بدهند. نمودار بعدی state machine هست و میگوید چه حالاتی وجود دارد و چه event هایی باعث تغییر حالات میشوند. میگوییم یک ساب سیستم یا یک کلاس یا یک instance کلاس یا یک کامپوننت از وقتی ایجاد میشود تا موقعی که میمیرد چه حالاتی دارد و چه ایونت هایی باعث تغییر حالت میشود و در جریان انتقال از یک حالت به حالت دیگری چه اکشن هایی انجام میشود. اکشن کار اتمی هست یعنی انجام میشود در یک لحظه و تمام میشود اما در اکتیویتی زمان دارد در طی زمان هست هر اکتیویتی یک سری اکشن دارد ولی داخل اکشن مشخص نیست. هر جا چنگ دیدی یعنی خودش این یک اکتیویتی هست و جزئیات آن در دیاگرام اکتیویتی هست. روی استیت ها اکتیویتی و روی

انتقال ها اکشن گذاشته میشود تا نشان بدهیم داخل یک حالت چه اکیتیوتی و داخل یک انتقال چه اکشنی انجام میشود. برای کلاس هم میشود state machine تعریف کرد که بهش object lifecycle diagram گفته میشود که نشان میدهد از زمان تولد تا مرگ چه حالاتی براش شکل میگیرد. ابجکت ها میتواند وابسته به حالت باشد و حالت در رفتار موثر هست. اگر رفتار وابسته به حالت داریم خوبه از این نمودار استفاده کنیم و گرنه مناسب نیست ما خیلی از سرویس ها بدون حالت هست یعنی فقط سرویس میدهند و رفتار وابسته به حالت ندارند کلاس منظورم هست. interaction دیاگرام: که تعامل بین ابجکت ها را نشان میدهند، مهم ترین آنها sequence diagram هست، یا نمودار ترتیب، یا توالی که هر ابجکت یک خط زمانی دارد و انتقال پیغام بین ابجکت ها نشان داده میشود از کجا متوجه توالی میشویم؟ اون پیغام هایی که بالاتر هستند ترتیب بالاتری دارند و اولویت بیشتری دارند اون پیکان ها. زمان از پایین به بالا اضافه میشود. هر یوزکیس به یک sequence diagram تبدیل میشود، یعنی نشان میدهیم هر یوزکیس چگونه با تعامل ابجکت ها در زمان اجرا محقق میشود. کمان منتهی به ابجکت یعنی داریم ابجکت میسازیم. شروع کننده ابجکت توالی انتقال پیام ایجاد شدن ابجکت درگیر شدن و اتمام آنها نشان داده میشود. داخل اون مستطیل دراز نشان میدهیم توی اون زمان ابجکت کدام متدش در حال اجرا هست. communication diagram: ابجکت های درگیر را نشان میدهد و بین ابجکت ها لینک شده است در زمان اجرا روابط را مشخص میکند رو لینک ها پیغام ها هست ترتیب انتقال پیغام از طریق شماره توالی نشان داده شده است. timing diagram: نشان میدهیم چه ابجکت هایی وجود دارد محور زمان دارد و یک سناریو کامل اجرا میشود در تعامل ابجکت ها با هم و حالات هر کدام مشخص هست و سناریو تعریف میکنیم تا بگیم این ابجکت ها چگونه با هم کار میکنند و پیغام میفرستند. هدفش این هست که یک سناریو رفتاری تعریف کنیم تا نشان بدهیم ابجکت ها با هم تعامل دارند و درگیر میشوند در زمان اجرا در یک سناریو و محدودیت های زمانی را تصریح میکنیم. نمودار را compact میکند. Interaction overview diagram یک اکیتیوتی دیاگرام هست که نود هاش interaction diagram هستند، که نشان میدهد که رفتار های درشتانه سیستمی به چه ترتیبی اجرا میشوند. برای بیان ترتیب مجاز اجرای یوزکیس ها هر یوزکیس با sequence diagram نشان داده میشود و ترتیب مجاز هم توسط این دیاگرام. هر کدام از نود های آن خودش یک sequence diagram هست. قبلا ترتیب با پیش شرط پس شرط بود واسه یک یوزکیس ولی الان با این دیاگرام نشان میدهیم. خود این interaction diagram هست که میتوانیم تو در تو باشیم.

جلسه چهارم:

USDP: فرآیند یکپارچه ایجاد نرم افزار که نسخه ساده شده RUP هست. نسل سومی و سنگین و کارا هست. باید سفارشی و تا حد خوبی کوچک شود تا کار های زائد از بین برود. متدلوژی یعنی چی؟ یک چهارچوب هست، انواع اقسام رویه های کاری میبینید رویه عملی یا کاری این روش ها از درشتانه تا ریز دانه داخل آنها هست. چهارچوبی که این حجم وسیع ابزار را نشان میدهد که میگوید ما از چه ابزار هایی باید در کجا استفاده کنیم، و متدلوژی سنگین وزنی نیاز هست تا بتواند تیم های مختلف را مدیریت کند، بتواند تضمین کند انتقال اطلاعات بین اعضا و بین تیم ها درست هست و ثبت اطلاعات به درستی و با پیچیدگی مناسب انجام میشود و در ذهن افراد درگیر هست که ما داریم پیشرفت میکنیم و موفق میشویم یک چهارچوبی هست برای اعمال رویه های مختلف مهندسی نرم افزاری برای سیستم هایی که نرم افزار در آنها نقش اصلی را ایفا میکند و هدف ما ساخت نرم افزار هست. یک متدلوژی دارد اون رویه های مختلف نرم افزاری را دور هم جمع میکند بهش ترتیب و التزام میدهد برای اینکه بدانیم از کدام practice چگونه و در کجای پروژه و با چه هدفی استفاده کنیم. متدلوژی با جزئیات کافی هست. میتوانیم از بعضی ابزار ها استفاده بکنیم یا نکنیم به این درجه آزادی میگویند. تا حد خوبی ملموس هستند. مشکل اصلی بلاتکلیفی هست مثلا ما این همه روش میدانیم ولی کجا باید از اینها استفاده کنیم متدلوژی اینجا مشخص میکند. و با بقیه

رویه ها و فعالیت ها مچ و جفت میشود را مشخص میکند در متدولوژی. متدولوژی یک چهارچوبی هست برای کنار هم جمع کردن practice ها التزام و ترتیب دادن به آنها و دونستن تکلیف مهندسی نرم افزار. Development در اینجا معنی تکوین را میدهد نه بسط یا گسترش چون ما چیزی نداریم که توسعه بدهیم معانی مختلف دارد پس ما اینجا تکوین میکنیم یک پروژه را به صورت تدریجی. متدلوژی یعنی کل چرخه عمر ایجاد یک نرم افزار را پوشش میدهد. بعضی متدولوژی ها هستند که بعضی قسمت ها را پوشش نمیدهند. مثلا بخش maintenance را بعضی ها پوشش نمیدهند کلا سه مرحله اصلی داریم در چرخه عمر ایجاد یک نرم افزار definition و development و maintenance. در اولین قسمت سیستم مطلوب را تعریف در دومی میسازیم و در سومی مراقبت میکنیم سه فاز ترتیبی هستند. یک متدلوژی باید بگوید که یک روش درشتانه هست که چگونه شروع کن جلو برو و به محصول نهایی برس و بعد چطوری مراقبت کن. مراقبت مهم ترین این مرحله هست. امروزه روش توسعه نرم افزار evolutionary هستند یعنی مدام داری به نرم افزاری که ساختی قطعاتی اضافه میکنی، iterative incremental هستند. کار ها در maintenance و development شبیه به هم شده است. یعنی DV که میشه تکوین و operations با هم همپوشانی دارد چون دومی الان یعنی برنامه بر بستر سیستم نصب شده است. واسه همین رویکرد devops داریم. در هم تنیده هستند. متدلوژی ایجاد نرم افزار ۲ بخش دارد ۱. زبان مدل سازی ۲. فرآیند هدفش این هست که گردهم بیاورد مجموعه ای از practice و ابزار های مهندسی نرم افزار را و بهش انتظام بدهد تا ما بتوانیم یک نرم افزار بسازیم. زبان مدل سازی مجموعه ای است از قرار های مدل سازی هم شامل نحو هم معنا را در بر دارد. زبان بصری لزوما نیست بصری UML هست ولی فقط ناظر به بصری نیست مثل نمودار بلکه مستندات متنی تولید میکنیم که برای این ها متدلوژی ها قالب مشخصی تولید میکنند مثلا توصیف یوزکیس با استفاده از یک جدول تعریف میشود و UML تعریف نکرده است این تعاریف را یا ساختار جدولی را. این ساختار جدولی قالب مشخص دارد. محصول باید توصیف کند یوزکیس را و همین جز زبان مدل سازی ما میشود. کد یکی از محصولات متدولوژی هست در جریان استفاده از متدلوژی تعداد زیادی محصول تولید میشود مثل دیاگرام ها یا مستندات متنی برای همین ها هم قالب تعریف میکنند همین هم زبان مدل سازی هست. کد هم زبان مدل سازی هست چون ما زبان ماشین پیچیده را بازنمایی میکنیم به زبان انسان به لاجیک تمرکز میکنیم بدون تمرکز روی پیچیدگی ها. زبان برنامه سازی یک زبان مدل سازی هست. همه قوانینی هم که روی این محصولات تولید شده نظارت دارند جز زبان مدل سازی هست. بخش فرآیند: شامل ۳ تا P هست اولی process unit یعنی همان اکتیویتی یا فعالیت ها یعنی چه کار هایی باید در متدولوژی انجام شود، فرآیند یکی از کارهاش این هست که بگوید کار های که در زمان اجرا انجام بشود چی هستند ترتیب چی هست و چه ارتباطی با هم دارند. P دوم محصولات هستند یعنی product، artifact یک کامپوننت زمان اجرا نرم افزاری که در روی یک نود نشسته است. اینجا منظورمان از artifact هر محصولی هست که در زمان اجرای متدلوژی ساخته میشوند. P سوم people یا producer تولید کنندگان هستند که نقش های درگیر پروژه هستند اینها نقش هستند نه فرد. متدلوژی به نقش ها کار دارد و تکلیف فرد هم مشخص کند مثلا در هر تیم یک فرد باید این نقش را داشته باشد یا دو تا نقش را نمیتواند یک فرد داشته باشد و افراد مختلف باشد چون نمایندگان منافع متضاد هستند که در مذاکره بین اینها پروژه شکل بگیرد. باید مسئله تیم ها را هم مشخص کند که چه تیم هایی داریم متشکل از چه نقش هایی و این تیم ها چگونه باید با هم دیگر تعامل میکنند. مدیریت افراد و مدیریت تیم ها یکی از دغدغه بخش فرآیندی هست. یک سری محصولات میانی داریم که ساخته میشوند تا کمک کنند به محصول نهایی برسیم و باید بگوید هر محصول بر اساس کدوم بخش زبان مدل سازی باید ساخته بشوند و قواعد نحوی و معنایی را رعایت کنند. کدوم از این ۳ تا مهم هست؟ product یا محصول چون فعالیت و افراد را انجام میدهیم تا محصول ساخته شود محصولات محوری هستند ولی بخش اصلی فعالیت هست. در توصیف متدولوژی ۳ تا رویکرد میبینیم: process unit oriented و product oriented و people oriented. وقتی توصیف اولی رو بگیم یعنی p اول محور هست چه کار

هایی با چه ترتیبی و در هر کار چه محصولی تولید و اپدیت و چه افراد و نقش هایی درگیر هستند. جای oriented میایم centric هم میگیم. Process unit چتری هست یعنی میگوید اول چه کاری بعد چه کاری و به صورت موازی هست. مثل product grooming در متدلوژی اسکرام، در process oriented توضیح میدهد و اینکه چه افرادی درگیر هستند و چه محصولاتی تولید میشود فرع هست. در هر کدام از اینها میگیم کدام p نقش اصلی هست و مثلا در نقش ها هر نقش در چه کار هایی درگیر هست و برای تولید محصولاتی تولید میشود. ما روی process unit تمرکز داریم بعد میگیم چه نقش هایی درگیر هستن و چیا تولید میشوند. ۳ تا p هست و یک مجموعه ای از معیار های برای پایش و اندازه گیری محصولات و فعالیت های پروژه، یعنی محصولات درست و دقیق و ناسازگاری نداشته باشند. کامل بودن هم مهم هست. در حدی باشد محصول که همه چی را داشته باشد در حدی که ابهام نداشته باشد. متدلوژی خودش framework هست. دقت کن framework ها از نظر سطح abstraction متفاوت هستند. رویکرد agile میگوید برگردیم به ریشه ها و بپردازیم به اصل قضیه و کدینگ چرا اینقدر وقت سر نقاشی و نمودار و مستندات میگذاریم. Up ها سه عضو هستند در خانواده EUP برای شرکت های بزرگ و سازمان ها. UP برای پروژه های کوچک تر و RUP برای پروژه های rational. بعضی متدلوژی ها suitability فیلتر دارند و با پروژه ما خودشون رو تناسب میدن ببینند مناسب هستند یا نه. Framework چون خیلی زیاد abstract هست و خیلی چیز ها را نمیگوید reusable هست واسه پروژه های مختلف. مدل سازی برای scalability کمک میکند ولی خیلی از agile ها اهلش نیستند. UML با UP یک ریشه دارد. UDSP به اصطلاح use case driven هست، اگر بگیم یوزکیس بیسد یا requirement based یعنی نیازمندی ها اول تعریف میشوند بعد تحلیل و طراحی بعد پیاده سازی و تست. برای نیازمندی های بعدی باید از ماتریس استفاده کنیم. که معلوم بشود محصولات چگونه تریس میشوند به نیازمندی ها. در یوزکیس بیسد ممکن هست فاصله پیش بیاد بین نیازمندی و محصولاتی که بر اساس نیازمندی ساخته میشود. بخاطر نقش محوری یوزکیس میگوییم use case driven یعنی بر اساس اون تحلیل و طراحی و پیاده سازی میشود. Use case driven نوعی خاصی از requirement based هستند. agile ها و نسل سوم requirement based و use case driven هستند. architecture centric: یعنی اینکه مبتنی بر این هست که معماری سیستم اول بیاد بیرون بعد بر اساس آن جلو برود منظور این هست که از همان ابتدا تعریف و مبنای بقیه کار ها قرار میگیرد قدیم تو فاز تحلیل کاری نداشتند و جز قلمرو جواب و ساختار سطح بالا بود و از طراحی به بعد بود. تحلیل ۲ فاز هست مقدماتی و تفضیلی و طراحی هم ۲ فاز هست طراحی معماری و طراحی تفضیلی. در پایان هر iteration یک محصولی ساخته میشود یک incremental و به بقیه اضافه میشود. 4 فاز در UP وجود دارد، ترتیبی هستند و در انتها یک releaseاز نرم افزار در محیط کاربر قرار میگیرد. Inception یعنی آغاز. Elaboration یعنی تفضیل construction ساخت transition انتقال. در فاز اول دید کلی نسبت به پروژه را داریم و مشخصات پروژه را کسب میکنیم و ریسک ها را میسنجیم و راجب امکان پذیری پروژه و جلو رفتن آن صحبت میکنیم در فاز دوم جزئیات سیستم کامل بدست میاد و تحلیل تفضیلی هست برخلاف فاز اول که تحلیل مقدماتی هست. تا ۸۰ درصد نیازمندی ها و طراحی معماری هم اینجا صورت میگیرد یعنی وارد طراحی هم میشویم. در فاز سوم طراحی تفضیلی و implementation و تست را انجام میدهیم و در جریان transition میایم نسخه تولید شده در فاز سوم را در بستر سیستم کاربر قرار میدهیم وقتی که نرم افزار در سیستم کاربر نشست و کارش را شروع کرد از آنجا باید maintenance شروع شود. خودش ساپورت نمیکند در up.

جلسه پنجم:

هر فاز یک سری مراحل دارند که این فاز ها را از هم جدا میکنند که به اینها milestone یا فرسنگ شمار گفته میشود. فعالیت ها یا محصولاتی که نشان دهنده اتمام فاز هستند که هدف آنها این هست که این فاز تمام شده است این میتواند یک محصول یا یک وضعیت یا حالت باشد یا یک جلسه باشد که وضعیت یا حالت را ترجیح نمیدهند بلکه میگویند اتمام فاز باید تحویل محصول یا یک جلسه باشد البته محصول رایج تر هست چون شهود بیشتری دارد. در مورد فاز اول تحلیل مقدماتی هست سنجش ریسک و شناسایی اول پروژه انجام میدهیم بعد امکان پذیری میکنیم ببینیم ادامه بدهیم یا نه. این فاز را مدیران ارشد پروژه و با تجربه انجام میدهند یعنی افرادی را میگذاریم که سریع در ۴ هفته حداکثر به نتیجه برسند در همه جهات. این را نمیسپارند به آدم تازه کار. قبل از اینکه وارد فاز دوم بشیم یک فرسنگ شماری به نام vision داریم که سند چشم انداز هست که باید اینقدر اطلاعات کسب کرده باشیم که بتوانیم چشم انداز را تهیه کنیم و این فاز را ببندیم. مهم ترین آن سند چشم انداز هست محصولات و نتایج و شرایط دیگری هم هستند که مهم ترین آن همین سند چشم انداز هست. قدیم میگفتند فرسنگ شمار باید مجموعه ای از شروط باشد که الان قابل قبول نیست و الان متناظر میکنند با یک محصول که اگر اون تهیه شد که هیچ اگر تهیه نشد هنوز کارمان تمام نشده است. در فاز دوم باید تحلیل تفضیلی بکنیم تمام جوانب و ریسک ها را بشناسیم و برطرف کنیم و معماری سیستم را مشخص و تثبیت کنیم. معماری ۲ وجه دارد یکی ساختار سطح بالا که هم در قلمرو مسئله هم قلمرو جواب هست وقتی در قلمرو مسئله هستیم یعنی زیر سیستم هایی که سیستم اصلی را میسازند و در هر پیاده سازی هست، این میشود معماری در تحلیل یا قلمرو مسئله ولی بیشتر در قلمرو جواب یا طراحی هست یک بخشی این هست که بگیم کامپوننت های درشتانه چی هستند و چگونه با هم ارتباط دارند، و چگونه روی بستر قرار میگیرند اینها همه در طراحی معماری هستند در قلمرو جواب. بعضی از کامپوننت ها متناظر میشوند با همان کامپوننت ها در فاز تحلیل اما بعضی از آنها نه مختص قلمرو جواب هستند مثل ui و دیتابیس، لایه ارتباط با ماشین یا شبکه که در کامپوننت دیاگرام میان. در طراحی معماری کامپوننت دیاگرام و دیپلویمنت دیاگرام هستند ساختار سطح بالا سیستم در قلمرو جواب یک وجه معماری هست و وجه دیگر مشخصات و ویژگی های قلمرو جواب هست مثل زبان برنامه نویسی DB و ویژگی های شبکه و ...، بعضی از اینها که مربوط به بستر هست مثل پروتکول ها در دیپلویمنت دیاگرام دیده میشود ولی بعضی ها نه و جداگانه باید مدل ثبت میشود در مستندات متنی، انتخاب و تعیین اینا و ویژگی های فنی در قلمرو جواب معماری هستند که یک وجه دیگر هستند. هر چی که مربوط به وجوه فنی باشد بستر چطوری هست و ساختار چطوری هست نمیتوانیم در دیپلویمنت دیاگرام بیاوریم هر چیزی که مربوط به وجوه فنی قلمرو جواب باشد این هم ناظر بر ابزار هایی هست که استفاده خواهید کرد برای ایجاد هم اینکه محیط نهایی که نرم افزار قرار میگیرد دارای چه ویژگی هایی هست باید تعیین تکلیف کنیم در قسمت راه حل نرم افزاری یا solution و قلمرو جواب هستند. در پایانه فاز دوم همه ویژگی های قلمرو جواب که از چه ویژگی هایی استفاده میکنیم، و همه تصمیمات مربوط به معماری مثل ساختار سطح بالای تحلیل و طراحی و ویژگی های قلمرو جواب مشخص شده و تثبیت شده است چطوری یک نسخه اول از نرم افزار داریم که همه این ویژگی ها را دارد و آماده است قابلیت پردازشی و سرویس و یوزکیس ها را بپذیرد فقط کد زیر ساخت شده است شمای کلی دیتابیس به طور کامل مشخص نیست، ui تا حدود خوبی پیاده سازی شده است و در نسخه اول یا executable architectural baseline که یک prototype اولیه هست اخذ شده است، یعنی در آن ویژگی معماری تثبیت شده است به اصطلاح یک prototype تکاملی هست چون گفتیم prototype دو نوع هست دور ریختنی و تکاملی یا evolutionary که دومی را میسازیم تا به محصول نهایی برسد اولی را میسازیم برای یک وجه مثلا برای ریسک یا یک نیازمندی را خوب نمیشناسیم یک چیز میسازیم به کاربر نشون میدهیم میگیم این هست چیزی که میخوای؟ یا همین سرویس هست یا میخوای تخمین بزنی یک یوزکیس پیاده سازیش چه قدر طول میکشد یکی میزنی تا ببینی چه قدر پیچیدگی دارد و چه قدر طول میکشد تا جایی میزنیم که بتوانیم تخمین بزنیم یا میخوایم از یک تکنولوژی استفاده کنیم ولی مطمئن نیستیم جواب میدهد یا نه از یک prototype دور ریختنی استفاده میکنیم ببینیم جواب میدهد یا نه، یا prototype معماری میزنیم یعنی مثلا چند تا معماری داریم ببینیم کدوم برای سیستم بهتر جواب میدهد همان را میزنیم، در prototype اصلا

تست نمیشود و اصلا از لحاظ کیفیت نباید تعریفی داشته باشد و نباید اینها را به عنوان محصول نهایی استفاده کنیم مثال دود و آینه. حامد اگه تا اینجا رسیدی کیرم دهنت. اپسیلونی از کیفیت نباید زد. Anti pattern ها نشان میدهد چه چیزی در صنعت رایج هستند ولی باعث ضرر به کیفیت میشوند.

در فاز دوم مهم ترین مسئله کیفیت همین قضیه معماری هست. بعد در فاز سوم اون executable baseline را که در مرحله قبلی ساختیم تکامل میدهیم و به سیستم آماده تحویل به مشتری تحویل میدهیم در فاز سوم تحلیل architectural کم رنگ ولی طراحی تفضیلی به شدت پر رنگ هست، معماری را تثبیت میکنیم در فاز دوم و زیاد از اینجا خبری نیست چون اگر معماری تغییر کند خیلی پر هزینه هست در این فاز سوم. مثلا ui باعث کل تغییر در سیستم میشود. تغییر به شکل صفر نیست بلاخره تغییرات هست ولی حداقل اندک هست. در فاز سوم یوزکیس ها را محقق میکنیم و در آخر آن ما یک محصول آماده ارائه به کاربر داریم، که initial capability، چرا میگویند قابلیت اولیه؟ چون به عنوان محصول ترخیصی الان تعیین شده است و اون must have های دید کاربر یا به اصطلاح MRF ها در آن دیده شده است، یعنی اونایی که حتما از دید کاربر باشد امده هست و سایر ویژگی ها یا در release بعدی هست یا به صورت perfected maintenance صورت میگیرد. در فاز چهارم این محصول را release میکنیم و فرسنگ شمار آن نرم افزار در بستر کاربر نشسته است و ثبات پیدا کرده است. دقت کن در فاز چهارم مشکلات اولیه باید حل بشود و پیکربندی ها باید صورت بگیرد و سیستم باید نصب شود. testing های مختلف در فاز چهارم صورت میگیرد. در بستر کاربر، acceptance test صورت میگیرد. تا پذیرش نهایی سیستم صورت بگیرد. و این مسائل مهم هست و ممکن است در محیط ایجاد بعضی از آزمون ها را انجام بدهیم برای اعتبار سنجی چه فرقی هست بین verification و validation: اولی صحت سنجی به این معنی هست که سیستم درست کار میکند یا نه و testing ناظر بر این هست. دومی میگوید اعتبار سنجی یعنی سیستم معتبر هست یا نه یعنی همان چیزی هست که کاربر میخواهد هست یا نه. Agile, up گفته اند که اصلا نمیتوان کیفیت را زد. یک پارامتر چهارم داریم تحت عنوان scope، یعنی حوزه نیازمندی ها و با این بازی میکنند جا کیفیت یعنی وقتی وقت کم شد scope رو میزنیم و وقتی وقت کم اومد جا کیفیت از نیازمندی ها میزنیم یعنی نیازمندی ها را رتبه بندی میکنیم بر اساس ارزش، به ۴ دسته ، به این ۴ دسته چی میگن؟ قواعد مسکو، بدون اون ۲ تا oo. ولی در واقع اون m, s, w, c را میخواهیم در یادمان باشد، m مخفف must have هست، یعنی نیازمندی که حتما باید باشد و گرنه پروژه شکست میخورد. S مخفف should have هست حداکثر تلاشمون رو میکنیم باشند ولی اگه نبودند پروژه شکست نمیخورد. C مخفف could have هست یعنی میتوانند باشد ولی تزئینی هستند یعنی در صورتی که پول و وقت زیاد بیاد میتوانیم انجام بدهیم و گرنه پیاده سازی نمیکنیم. won't have، اینها را پیاده سازی نمیکنیم ولی دور هم نمیریزیم، چون شاید won't have بشود must have، ممکن هست این اولویت ها بهم تبدیل شود بعد اگر پول و وقت کم آوردیم اینها را میزنیم از اولویت پایینی ها. از must have نمیتوانیم حذف کنیم. این اولویت کمک میکند برای تغییر scope. این پارامتر ۴ ام پروژه شد. پارامتر کیفیت قابل قربانی کردن نیست. بابت زدن کیفیت باید هزینه بدیم اگر پرداخت نکنیم بدهی فنی را، یعنی اگر وقت نگذاریم برای refactoring مشکلات ما شدید تر میشود و بدتر میشود. که میشود technical debt که میشود بدهی. در agile از کیفیت کوتاه نمیایم. در XP یک استراتژی خیلی سخت داریم که کد استاندارد هست تا افراد بتوانند کد همدیگر را بخوانند. وقتی بخواهیم validation بکنیم باید acceptance test، user story ها را اجرا کنیم تا ببینیم آنچه که ثبت شده است به عنوان نیازمندی، تحقق نیازمندی صورت میگیرد یا نه البته امروزه به این شکل نیست و نیاز به عامل انسانی داریم. بعد از نصب باید انجام شود این validation حالا چه توسط خودش چه توسط نماینده اش، این acceptance test در مرحله آخر هست بعد از نصب در سیستم

کاربر. البته امروزه ما validation مستمر داریم و کاربر جز تیم ما هست، تیم ساپورت دست به کد نمیشود و فقط سیستم را بالا نگه میدارند یا به کاربر راهنمایی راجب سیستم میکنند کار های administration انجام میدهند و بک اپ میگیرند و لاگ میگیرند. Planning، maintenance هم باید انجام شود. در جریان maintenance، کیفیت میاد پایین. همه اینها در فاز چهارم هست. این متدلوژی USDP، فاز maintenance ندارد و فرض این است که برای اجرای آن از فعالیت های تکراری خودش باید انجام شود که حرف بی معنی هست و نمیشود و نیاز داریم به تمهیدات مستقل چون ایجاد با نگهداری فعالیت های مختلف هست ایجاد بر اساس plan ها هست ولی نگهداری، یک چیز interrupt driven هست و بر اساس نقشه ای نیست. scrum، بدرد interrupt driven نمیخورد چون spring goal ها را حداکثر تلاش میکنی تغییر نکند ولی در صورتی که maintenance یک کار غیر پلن هست و غیر پیش بینی هست، در فاز چهارم برنامه ریزی میکنیم برای نگهداری و یک متدولوژی برای آن تعریف میکنیم. فاز چهارم بیشتر از چند هفته نباید طول بکشد بعد از پایدار شدن سیستم این فاز اتمام میشود. پس در فاز اول هم vision هم scope هم business case در میاوریم، سومی یعنی توجیح اقتصادی پروژه و دومی هم میشود core requirements یعنی نیازمندی های هسته. در فاز دوم: تعریف از محصول و تحلیل تفضیلی در فضای قلمرو مسئله و معماری را طراحی میکنیم و یک محصول کلی میسازیم و یک پلن دقیق تر برای ایجاد و مستقر سازی در بستر کاربر را انجام میدهیم. در فاز سوم محصول نسخه اولیه که به عنوان executable baseline architectural ساختیم را به محصول نهایی میرسانیم و در فاز چهارم نصب میکنیم برای استفاده کاربر فاز چهارم شامل ساپورت maintenance آموزش و راهنمایی و انتقال هست. دقت کن در فاز اول ما یک دیدی از معماری بدست میاریم، برای سنجش ریسک تا حدودی در قلمرو جواب هم وارد میشویم ولی دقیق نیست و نمیتواند در بیاد و فاز دوم دیگر کامل هست و یک تغییرات ریزی در فاز سوم دارد.

جلسه ششم:

در هر فاز یک سری iteration داریم، رویکرد ما در UP، iterative incremental هست یعنی اینکه به صورت تکراری بعضی کار ها را انجام میدهیم و در انتهای هر کار یک بخش یا یک تیکه ساخته میشود و به سیستم اضافه میشود امروزه تمام متدلوژی های اعم از سنگین و agile ها این شکلی هستند. دقت کن اینها increment هستند نه release، release آن بخشی از نرم افزار هست که در سیستم کاربر مینشیند، که به این سیستم کاربر user environment یا production environment گفته میشود. چیز هایی که در انتهای هر iteration ساخته میشود لزوما در محیط کاربر نمیشیند بلکه لزوما ساخته و اضافه میشود به بخش های قبلی که گفتیم increment نام دارد، ولی release در فاز چهارم هست. در فاز اول یک prelim iteration داریم، کلا به iteration های فاز اول prelim گفته میشود در این ما لزوما تیکه نرم افزار ساخته نمیشود اصلا ایجاد نرم افزار نیست بلکه prototype میسازیم که دور ریختنی هست و تحلیل مقدماتی هست و برای سنجش ریسک هست. در فاز دوم واقعا نرم افزار میسازیم یعنی executable architectural baseline میسازیم که یک تیکه نرم افزار واقعا هست به این iterationها، architectural iteration گفته میشود، وظیفه آن مشخص کردن و تثبیت معماری سیستم هست. در فاز سوم به شدت نرم افزار تولید میکنیم و تعداد iteration ها زیاد هست و در انتهای هر کدام یک increment نرم افزاری میسازیم، به اینها development iteration گفته میشود. فاز چهارم هم بهش transition iteration گفته میشود. فاز اول و چهارم معمولا فقط یک یا دو iteration دارند. ولی دوم و سوم معمولا زیاد هست. ۵ تا فعالیت داریم که در هر iteration انجام میشود.

Requirements, analysis, design, implementation, test. در اولی یوزکیس را در میاوریم در آنالیز بخشی از سیستم که هدف گرفته شده را متمرکز و دقیق میشویم و در قلمرو مسئله هستیم و به پیاده سازی کاری نداریم میخواهیم بدانیم سیستم چی هست مدل سازی و معماری در قلمرو مسئله اینکه از چه ساب سیستم هایی تشکیل شده هست این ساب سیستم که همان پکیج ها هستند چه ارتباطی با همدیگر دارند و همچنین چه ساختار هایی یا چه کلاس هایی درون این پکیج ها هست، بعد به رفتار ابجکت ها میپردازیم که چطوری بهم پیغام بفرستند تا یوزکیس تحقق پیدا کند کلا هدف ما این هست که یوزکیس ها محقق شود. و در آنالیز میگیم چه کلاس هایی متمرکز بر این یوزکیس ها هستند تا آنها را محقق کنند. در design دنبال این هستیم که این معماری در قلمرو مسئله چطوری در قلمرو جواب نمود پیدا میکنند و پکیج تحلیل چگونه به کامپوننت طراحی تبدیل میشود و از قلمرو جواب چه کامپوننت هایی بهش اضافه میشود ui ارتباط با ماشین و ساختار پلتفرم و db اینجا وارد میشوند. اینجا هم مبنا یوزکیس هست. و مجموعه ای از کلاس های طراحی و ارتباط آنها به همراه ابجکت ها در کنار هم قرار میگیرد تا یوزکیس محقق شود. بعد blue print یا نقشه بدست میاد بعد میدهیم به برنامه نویس و اون یوزکیس ها در نرم افزار محقق میشود. الان implementation and test با هم صورت میگیرد و محیط ها مبنا تست هست یعنی بر اساس تست جلو میرویم. ۳ ویژگی وجود دارد stored, reputable, ultimate یعنی تست ها باید تعریف بشوند که یک مجموعه ای ذخیره شده از آنها هست به صورت مکرر و اتوماتیک. Test driven این شکلی هست که اول تست کیس در میاوریم بعد کد میزنیم تا کد ما بتواند تست کیس ها را رد کند. به این ۵ تا mini waterfall گفته میشود. به این ۵ تا workflow گفته میشود در UP. البته در RUP و UP جدید به اینها discipline گفته میشود. در فاز اول تحلیل مقدماتی و نیازمندی ها تا حدی پیدا میشوند در فاز دوم تحلیل تفضیلی میکنیم و کل نیازمندی ها پیدا میشوند و طراحی معماری میکنیم. در فاز سوم روی پیاده سازی و تست صورت میگیرد و طراحی تفضیلی هست در فاز چهارم تست های validation و نصب نرم افزار را داریم. از فاز دوم به بعد در هر چرخش یک increment نرم افزار ساخته میشود و نصب میشود. فاز ها جاده و فعالیت های تکراری چرخ هستند. در طول فاز ها که جلو میرویم حجم کار از روی workflow های اولی به آخری ها حرکت میکنیم چون اول نیازمندی ها را بدست میاوریم بعد دیگه تمرکز میکنیم روی محقق. در فاز اول هر نیازمندی که بدست میاوریم میشود نیازمندی هسته، این ۸۰ درصد از نظر کاربر هست ولی از نظر تعداد ۲۰ درصد یوزکیس ها هست و به صورت درشتانه هستند. همه نیازمندی ها را همان اول نمیتوان در آورد و دیگر حد کامل و تعهد ما باشد غلط هست. خیلی زور بزنیم ۲۰ درصد هست. در ابتدا نیازمندی با جزئیات دقیق نمیتوان بیان کرد نیازمندی ها بروز یابنده هستند یعنی باید بروز کنند و ما بفهمیم بعد نیازمندی ها ممکن هست تغییر کنند و ثابت نیستند و ذاتا متغیر هستند، حتی اگر در آخر پیاده سازی باشد چون نیازمندی به مرور ساخته میشود. پروژه ها فقط یک ثابت دارند و اون هم لزوم تغییر هست و گرنه همه چیز همه چی تغییر میکند و ما باید بپذیریم. در آنالیز میفهمیم ریسک و پیچیدگی پروژه کجاست و از چه بخش هایی تشکیل شده و کجا مبهم هست و کدام قسمت ها در هم تنیده هستند. طراحی میکنیم design، وارد قلمرو جواب میشویم با هدف سنجش ریسک پروتو تایپ میزنیم تا ببینیم پیاده سازی یک بخش ریسکی چه قدر طول میکشد پروتو تایپ میزنیم تا ببینیم از یک تکنولوژی میتوانیم استفاده کنیم یا نه هدف امکان پذیری پروژه هست. implementation هم داریم که همان ساختن پروتو تایپ برای هدف سنجش ریسک بعضی از پروتو تایپ ناظر بر نیازمندی هستند یعنی وقتی پیاده سازی میکنیم و ابعادش مشخص میشود دقیقا با آن چیزی که کاربر میخواهد برابر هست یا نه جنبه های ریسکی انواع و اقسام هست. یکی از مهم ترین امکان پذیری financial feasibility یا امکان پذیری مالی هست یعنی اصلا سود دارد یا نه. از نظر دیگر tech feasibility داریم یعنی از لحاظ فنی کار میتواند صورت بگیرد. Schedule feasibility هست یعنی در زمانی که داریم کار قابل انجام هست یا نه. چهارمی operational feasibility هست یعنی در عمل کاربران میتوانند ازش در عمل استفاده کنند یا نه کاربرد پذیر هست یا نیست. application feasibility هم هست که همان چهارمی

هست اسم دیگرش، چون ممکن هست سیستم برای استفاده کاربران بسیار سخت باشد مثل ui پیچیده. سیستم های اطلاعاتی مثل سیستم های تجاری هم دو رویکرد دارند یک process intensification یعنی تمرکز روی پردازش داده ها باشد و داده ساده باشند یا data intensification که تمرکز روی داده ها باشد حجم آنها و پردازش ساده باشد سیستم های تجاری اطلاعاتی هستند از دومی هستند مثل سیستم آموزش که پردازش آنها ساده هست ولی حجم و تنوع داده خیلی بالا هست ولی تک یوزکیس ها خیلی ساده هست. امکان پذیری هایی دیگه هم هستند مثل امکان پذیری قانونی یا فرهنگی. خیلی از سیستم ها باید با تطابق خاصی داشته باشند از خیلی جنبه ها خیلی ملاحظات خاصی دارند. برای امکان پذیری مالی ۳ تا عدد رو باید پیدا کنیم، برای هر alternativeیی، یعنی alternative architecture ها که میشود همان راه حل ها ما هستند مثلا چطوری روی سیستم کاربر باشد مثلا web based باشد یا روی سیستم کاربر باشد و ... ، بعد برای این سیستم با این معماری طول عمر سیستم هم در نظر میگیرند مثلا عدد ۱۰ بعد امکان پذیری هر ۴ تا را روی آن میسنجیم در مالی ۳ تا عدد باید در بیاوریم ارزش خالص فعلی سیستم یعنی در طول ۱۰ سال عمر تمام هزینه ای که میکنیم استخراج میکنیم بعد دقت کن واحد پول باید به زمان حاضر باشد و یکسان باشد تا بفهمیم الان میصرفد یا نه، بهش present value گفته میشود چون ما انتظار برگشت این پول را داریم و پول بعد از ۱ سال باید ۲ برابر شود و سودمند ها را حساب میکنیم تا ارزش خالص فعلی بدست بیاد. عدد دوم return on investment هست یعنی روی سرمایه گذاری که میکنیم چه قدر سود برمیگردد و اگر کمتر از مقداری باشد مورد قبول نیست. سومین عدد نقطه سر به سر هست که سیستم دقیقا کی به سود دهی میرسد. ریسک ناظر بر feasibility ها هستند. در فاز اول تست نداریم و پروتو تایپ ها را تست نمیکنیم. در فاز دوم جدا وارد نیازمندی ها میشویم و در آخر این فاز نباید هیچ ریسکی نباشد حتی اگر وارد پیاده سازی نشده باشیم اینقدر وارد شدیم کاهش دادیم ریسک ها را. نیازمندی ها به ۸۰ درصد میرسند و بیشتر نمیشود تا آخر این فاز. ۲۰ درصد یوزکیس ها هم در فاز سوم بدست میاد کار های مربوط به ساپورت و بک اپ گیری و ریکاوری. در analysis تمام وجوه سیستم اینجا مشخص میشود و مدل سازی سنگین در قلمرو مسئله صورت میگیرد design را تا جایی جلو میبریم که بتوانیم یک معماری را تعیین و تثبیت کنیم و یوزکیس های خیلی ریسکی اینجا پیاده سازی میشوند تا خیالمان راحت شود در executable architectural baseline. طراحی اینجا بیشتر طراحی معماری هست به صورت کلی. در فاز سوم analysis, requirements کم رنگ و طراحی و پیاده سازی پر رنگ میشوند و سایر تغییرات در قبلی ها خیلی کم هست. طراحی معماری در آخر فاز دوم تمام هست و بعد از جزئی تغییرات خواهد داشت. معماری در implementation پیاده سازی میشود. دقت کن داخل executable architectural baseline هیچ فانکشنالیتی وجود ندارد و فقط یک سری فانکشنالیتی پایه هست مثل ارتباط با ماشین یا ارتباط با دیتا بیس که جنبه های معماری دارند و سرویس های انتظاری که کاربر دارد امکان پذیر نیست. پیاده سازی کوچک هست چون یوزکیسی نزدیم. تست هم داریم چون باید executable architectural baseline را واقعا تست کنیم. در فاز سوم تست هم سنگین هست و unit testing داریم و دقت کن اینجا دیگر خبری از نیازمندی نیست و نیازمندی جدیدی تعریف نمیشود در آنالیز و دیزاین هم خیلی کاری نداریم. یک سری دستکاری یا tunning ها هم داریم که وقتی در سیستم کاربر میشیند بتواند با بقیه اجزای سیستم کاربر درست کار کند. تنظیمات و دستکاری ها و رفع خطا ها نیازمندی پیاده سازی و مقداری طراحی هست تا کلاس ها تغییراتی کنند و سینک باشد پیاده سازی با تحلیل پس اون بخشی که از تحلیل باقی مانده هست مال همین چیزی هست که گفتیم چون اون تغییرات طراحی منعکس شوند در تحلیل. کار روی تست های مربوط به validation از فاز سوم آغاز میشود نه از فاز چهارم فاز چهارم اصل کارش هست. تکرار چطوری نمود پیدا میکنند؟ خوبی های تکراری بودن چیست؟ اگر همه increment ها بخواهند در آخر با هم یکی بشوند یک ریسک خیلی بزرگ هست که بهش integration risk گفته میشود پس بهتر هست به صورت مداوم با هم ترکیب شوند تا سیستم هایی سخت نشود که نتوانیم با هم ترکیب کنیم و واسط هایی خوبی تعریف

نشده باشد. مورد بعدی frequent executable release داریم یعنی به صورت زود به زود یک تیکه نرم افزاری ساخته میشود که در عمل release نمیشود ولی میتواند برود در محیط کاربر بشیند قابلیتش را دارد بیشتر هم internal هستند ولی release میشوند ولی delivery ندارند. ولی همینکه خرد به خرد سیستم را میسازیم قابلیت سنجش و کیفیت و تست دقیق داریم و میتوانیم به کاربر نشان دهیم. در iterative incremental، اصلا ما را حول میدهد به سمت قلمرو جواب یا راه حل و حول میدهد ما را به سمت ریسک ها تا همان اول انجامش بدهیم. این روش اصلا نمیگذارد ما فلج تحلیل بشویم و ما را مجبور میکند وارد قلمرو جواب بشویم تا ریسک ها را بشناسیم. پیشرفت بر اساس نرم افزار تولید شده میسنجیم چه قدر کد زدیم و کاری به مستندات نداریم و منظورمان پروتو تایپ های دور ریختنی نیست. احتمالش خیلی کم هست که بگیم در آخر فاز دوم یک ریسکی هست که دیگر نمیتوانیم کار را ادامه بدهیم چون وجوه مختلف و همه ریسک ها همان اول شناخته شده اند. کلا به این متدلوژی های agile و up میگوییم risk driven هستند یعنی کار ریسکی اولویت بالاتر دارد کلا روش های iterative incremental این ویژگی را دارد. اینکه معیار پیشرفت را همان نرم افزار موجود و اینکه چه قدر کد زدیم بگذاریم پایش پروژه آسان تر میشود و ریسک ها مشخص میشوند. مدیریت پروژه شامل ۳ تا فعالیت هست planning, scheduling monitoring and control: در اولی از سمت چپ یعنی تسک ها و وابستگی بین تسک ها و تخمین زمانی آنها را بدست بیاوریم چارت کشیدن مصداق برنامه ریزی هست. pert chart مصداق planning هست. project evaluation and review technique و تکنیک هست که چطوری این چارت و تحلیل مسیر را بکنیم و تخمین زمانی بزنیم در دومی باید تسک ها را ببریم روی تقویم و گنگ چارت ساخته میشود و مشخص میکنیم منابع به چه تسکی تخصیص پیدا کند و چه کسی این تسک را بزند با توجه به پیش نیاز های آن. کار سوم این هست که پیشرفت را مورد پایش قرار بدهیم و ببینیم پیشرفت میکنیم یا نه اگر گیر کردیم ابزار جدید بدهیم افراد جدید بدهیم و خیلی بهتر جلو میرویم. دقت کن به اون ۵ تا workflow، iteration planning و prepare release اضافه میشوند دومی را با پیاده سازی و تست را یکی میگیرند اولی نه جدا هست چون میگوید باید روی چه بخشی از سیستم متمرکز هستیم و قرار هست چیکار کنیم. در فاز اول امکان پذیری داریم مثل تکنیکال پروتو تایپ تا ببینیم جنبه های فنی یک فانکشنالیتی سیستم عملا با تکنولوژی در اختیار قابل پیاده سازی هست یا نه، و proof of concept prototyping میکنیم و آن چیزی که فکر میکنیم راجب سیستم درست هست را بسنجیم مخصوصا راجب نیازمندی ها تا به کاربر بگیم برداشت ما درست بوده است از این نرم افزار یا نه، business case میسازیم ببینیم توجیه اقتصادی دارد یا نه و سود مشتری را میرساند یا نه و باید به صورت مالی گفته شود و سایر کیفیت ها باید به صورت مالی ترجمه شود تا کاربر بر اساس آن تصمیم گیری کند. اون ۳ تا که گفتیم هم مال همینجاست از نظر اقتصادی. وظیفه مندی و قابلیت ها و ۲۰ درصد نیازمندی ها را مشخص میکنیم که همان ۸۰ درصد کاربر هست. ، با پیدا کردن ریسک ها. Human feasibility هم داریم که میتوانیم روی سمت مشتری حساب کنیم تا روی ما حمایت بکند؟ و یا ما آدم کافی برای انجام این پروژه رو داریم؟

جلسه هفتم:

باید توجیه اقتصادی برای سازمان در بیاد. به نیازمندی های هسته essential requirements هم گفته میشود. این کار را برای تعیین scope میکنیم. تا وقتی scope تمام نشود نمیتوانیم بگوییم که نیازمندی ها تمام هست ولی برای تخمین زدن ریسک ها و سنجش آن تا حدی جلو میرویم و همان مقدار هم کافی هست. هر فاز یک سری پس شرط دارد تا ببینیم این فاز کامل شده است یا نه. یکی از مهم ترین سوالات این هست که مقصود پروژه چیست و آیا میصرفه؟ آیا امکان پذیر هست؟ آیا سیستم را باید بخریم یا بسازیم از

صفر؟ آیا باید سیستم را از صفر بسازیم یا سیستم فعلی را بسط بدهیم؟ در هر تصمیمی باید هزینه را بسنجیم و تخمین بزنیم. آیا پروژه را ادامه بدهیم یا نه این مهم ترین سوال هست که آخر این فاز میپرسیم که پروژه رو ادامه بدهیم یا نه. Plan schedule همان تخمینی هست برای سنجش ریسک ها و زمان بندی ها و با آن کار را جلو میبریم که البته زیاد نمیتوان بهش اعتماد کرد. دقت کن امروزه خیلی مد هست که فقط برای release پیش رو تامین منابع میکنند تا ببینند کاربر اصلا سیستم را میخواهد یا نه و نه برای بعدی ها.

دقت کن یک product road map تعریف میکنیم تا مشخص کنیم پروژه نهایی را در چند release انجام میدهیم و زمان بندی دارد و مشخص میکند هر release چه معماری و ویژگی ها و نیازمندی ها را مورد هدف قرار میدهد و چه دسته از کاربران را مورد هدف قرار میدهد البته خیلی دقیق نیست و صرفا برای release بعدی مورد تایید هست. یعنی مثلا ما در فاز اول هستیم فقط واسه فاز دوم اون هم مال iteration های اول نه بقیه چون خیلی دقیق نیست واسه فاز سوم هم میشود ولی دقیق نیست اگر الان در فاز اول باشیم. ۳ تا p که باید اینها را به صورت زود به زود مورد بازبینی قرار بدهیم به صورت دوره ای در پروژه ایجاد نرم افزار: process, plan, product. در جلسه review ما plan, product رو مورد بررسی قرار میدهیم و در جلسات بعدی فرآیند رو مورد بازبینی انجام میدهیم یعنی محصول و ابزار و افراد که درگیر هستند و ناظر هستند روی پروژه را مورد بازبینی میکنیم. ما نباید صحت را فدای دقت کنیم یعنی precision را نباید فدای accuracy کنیم دقت کن ما تا یک جایی دنبال دقت هستیم فقط رنج تعریف میکنیم بعد دنبال صحت یا دقت هستیم. Velocity اینکه هر تیمی در هر sprint چه قدر کار میتواند بکند این هم با رنج هست. ما نمیتوانیم پلن ۱۰۰ درصد دقیق بدست بیاوریم و وقت میرود و همه کار ها نمیتواند دقیقا طبق پلن جلو برود و این کار هم غیر ممکن هست و ثابت نیست و تاریخ مصرف دارد در رویکرد های چابک در هر iteration یا آخر هفته برنامه را باز بینی میکنیم و مشکلات را زودتر متوجه میشویم و پلن خیلی مهم هست. دقت کن ما فرآیند یادگیری داشته باشیم تا پلن بهتر شود و بر اساس محصولات ملموس باشد تا بتوانیم محصولاتی بدهیم که قابل لمس باشد برای milestone ها. چه محصولاتی و چه شرط هایی باید ارضا شود تا فاز اول تمام شود: ۱. همه ذی نفع ها باید روی اهداف پروژه توافق کرده باشند، شامل اعضای تیم نرم افزار و سمت مشتری ذی نفع شامل هر کسی که در پروژه به نوعی دخیل هست و تاثیر میگذارد محصولی که نشان میدهد همه اعضا توافق کردند سند vision هست که در آن چشم انداز نیازمندی های اصلی و فیچر ها و محدودیت ها را میگوییم. منظور از نیازمندی functional requirements هست و منظور از فیچر non functional requirements هست. محدودیت های کاربر هم جز همین نان ها هستند مثلا حتما با زبان جاوا باشد. Constraint همان نیازمندی غیر وظیفه ای هست که از سمت کاربران هست و حتما باید لحاظ شوند و بعضی از اینها باعث ریسک هستند و میتوانند خیلی پروژه را پیچیده کنند مثل robustness یعنی سیستم خراب شد نخوابد. Reliable باشد یعنی دیر به دیر دچار مشکل شود همه اینها نیازمندی های غیر وظیفه ای هستند. بسیاری از اینها ناظر بر محیط ایجاد هستند مثلا چه نیروهایی لازم داریم چه قدر وقت داریم چه ابزار هایی داریم تازه ممکن هست به ما اجبار هم شوند و مال محصول نیست بعضی از غیر وظیفه ای ها مال محیط کاربر هست که مثلا پلتفرم حتما ویندوز باشد و ما نمیتوانیم خودمان معرفی کنیم دقت کن اونایی که کاربر اجبار میکنه باید جز قلمرو مسئله باشه. در قلمرو مسئله باید محدودیت های قلمرو جواب یا راه حل یا پیاده سازی را سریع بشناسیم چون دست ما بسته هست. دقت کن مکانیزم ها را نباید داخل یوزکیس بیاوریم فقط باید به صورت کلی بگیم و اینها دست کاربر هست، پس محدودیت های کاربر باید جز قلمرو مسئله باشد. بعضی ها معتقدند که عوامل فیزیکی و قلمرو جواب را وارد قلمرو مسئله نکنیم مثل همین محدودیت های کاربر. بعد باید scope را در بیاوریم و با ذی نفعان به توافق رسیدیم همچنین نیازمندی های کلیدی هم تولید شدند و به کاربر به توافق رسیدیم خروجی scope میشود ۱۰ تا ۲۰ درصد یوزکیس ها هست تا بتوانیم برای مدیریت ریسک بگوییم این تا احتمال بالایی قابل انجام هست و خروجی

نیازمندی های کلیدی میشود گلاسوری پروژه. Won't have ها در تعیین مرز سیستم بسیار مهم هست. گلاسوری مجموع واژه هایی هست که مربوط به قلمرو مسئله هست و نسخه اولیه اینجا و بعدا کامل میشود و فهم ما از مفاهیم مختلف مسئله تعریف میکند و میشود زبان مشترک ما با کاربر مثلا کاری که کاربر میخواهد و بر اساس آن با کاربر استخراج اطلاعات میکنیم و مصاحبه میکنیم. اهمیت گلاسوری اینجاست که ما وقتی بخواهیم مدل سازی کنیم باید زبان واحد داشته باشیم مثلا یک نفر به کارت بانکی نمیتواند بگوید کارت بعد یکی دیگه بگوید کارت اعتباری بعد هیچکسی هم متوجه نشود این کدوم کارت هست. پس در گلاسوری ما اسامی مختلف داریم که همه آنها یک نماینده دارند که ما از هر کلمه ای خواستیم استفاده کنیم نماینده آن را میاوریم تا زبان ما واحد باشد و رفع ابهام و تعیین و تکلیف هم نام ها هست و تعیین و تکلیف اسامی یکسان با معانی مختلف هست مثلا به سند ماشین و خونه نمیتوان گفت جفتش سند و گلاسوری باید اسم خاص برای هر کدام استفاده کنند تا ابهام پیدا نشود. گلاسوری منبع استخراج کلاس های ما هست. چپی ها شرط ها هستند. جدول پایین خلاصه همین حرفاست. یک سری تخمین های اولیه بر اساس نظر کاربر باید زده شود تا project plan و زمان بندی ها در بیاد. Business case باید در بیاد و کاربر تایید کند تا توجیه اقتصادی و سود دهی آن و معماری مختلف کاندید جانشین و امکان سنجی ها برای آن صورت میگیرد و حتما باید یک کاندید و معماری یا راه حل با ریسک پایین در بیاد که بگیم امکان پذیر هست بر اساس تمام محدودیت ها تا یک راه حل بدست بیاد برای این معمولا به شکل deployment diagram بدست میاد برای هر معماری که داریم به عنوان کاندید و جانشین مثلا در این معماری دیتابیس این شکلی هست و واسه هر معماری باید business case تعریف شود. خروجی ریسک های احتمالی هم باید بدست بیاد در این فاز و در یک سند باشد با پروتوتایپ مشخص میشود و برای ریسک پروتوتایپ میزنیم ریسک فنی و زمانی و پیچیدگی و امکان پذیری میزنیم و پروتوتایپ سنگین میزنیم.

Conditions of satisfaction	Deliverable
The stakeholders have agreed on the project objectives	A vision document that states the project's main requirements, features, and constraints
System scope has been defined and agreed on with the stakeholders	An initial use case model (only about 10% to 20% complete)
Key requirements have been captured and agreed on with the stakeholders	A project glossary
Cost and schedule estimates have been agreed on with the stakeholders	An initial project plan
A business case has been raised by the project manager	A business case
The project manager has performed a risk assessment	A risk assessment document or database
Feasibility has been confirmed through technical studies and/or prototyping	One or more throwaway prototypes
An architecture has been outlined	An initial architecture document

نکات ترکیبی از جلالیان:

این جریان می تواند پی در پی و متوالی(sequential) (، انشعابی یا شاخه ای(branched) (و یا همزمان concurrent) (باشد.

جلسه هشتم:

سند vision در نهایت حکم میکند که ما میتوانیم پروژه را ادامه بدهیم یا نه. مهم ترین feasibility از لحاظ مالی هست ریسک ها را هم باید شناسایی کنیم و پروتوتایپ میسازیم تا نقاط ریسک مشخص شود و همچنین ببینیم یک solution قابل حل هست مثلا وب بیس بر اساس فلان تکنولوژی مثلا با پروتوتایپ اینها را مشخص میکنیم. ما باید دنبال این باشیم که امکان پذیر و شدنی اینها و پیاده سازی نیازمندی ها بکنیم. استخراج نیازمندی ها هم در همین فاز صورت میگیرد با دقت خوبی. از اون ۵ تا کار جریان کاری: implementation and test در حد پروتوتایپ هست چون فقط پروتوتایپ داریم که نیاز به تست ندارد، در مورد ۳ تای اولی هر کدام جلو میروند تا جایی که بتوانیم راجب ریسک حکم بکنیم فقط هم تا همونجا بیشتر هم نه. در مورد analysis design هم برای شناخت سیستم نهایت برای تشخیص ریسک و تعیین اینکه یک معماری امکان پذیر قابل پیاده سازی وجود دارد با اطمینان بالا. طراحی تفضیلی بخاطر تشخیص ریسک در حد کم. طراحی معماری به همراه تحلیل تفضیلی هم مطرح هست برای شناسایی ویژگی ها. دقت کن vision از روی بقیه محصولات بدست میان این انگار جمع بندی همه چیز هایی هست که بدست آوردیم. Artifact نمود فیزیکی کامپوننت های زمان اجرا هست مثل file dll هر محصولی که در جریان اجرای پروژه ساخته میشود هم میگویند. فاز دوم: راه حل کلا در میاد و در قالب یک محصول در بیاد و معماری تثبیت میشود و محصول آن executable architectural baseline هست که یک نسخه نرم افزاری هست که تصمیمات معماری و ... در آن اتخاذ و تثبیت شده هست در این فاز تکلیف ریسک را یک سره میکنیم با همین نرم افزار و جنبه های مختلف آن را در میاوریم و ریسک کلا حل میشود و با جزئیات دقیق درست هست که ما کد زدیم یوزکیس ها را فقط جز اونا که ریسکی هستند پیاده سازی نمیکنیم و مال فاز بعدی هستند، ui و دیتابیس تا حد خوبی شکل گرفته و بر اساس یوزکیس ها و باید مطمئن شویم چیزی که مشتری از ui میخواد دقیقا هست پس باید تا حد خوبی پیاده شده باشد. User manual ها را در فاز دوم میسازیم نه فاز چهارم و دقت کن کلا هدفمان این هست که usability را بسنجیم. لایه های مختلف برای ارتباط با ماشین هم ساخته میشود و روی کیفیت آن تاکید داریم و quality attribute تعریف میکنیم و بر اساس رقم هست دو تاش هست با فرض انجام تست خوب نرخ پیدا کردن خطا یکی از ویژگی های کیفیت هست و اگر زیاد باشد یعنی کیفیت پایین هست، یکی دیگر چگالی پذیرش هست یعنی در چه حجم کدی چه قدر خطا داریم مثلا در ۱۰ خط خطا زیاد باشد خوب بد هست. عدم توجه به کیفیت iterative incremental را از بین میبرد. ۸۰ درصد یوزکیس ها در میاد و نیازمندی های غیر وظیفه ای شناخته میشوند یک پلن دقیق برای فاز سوم و یک پلن کمی ضعیف تر برای فاز چهارم بعد باید پروپوزال میدهیم کلا ۲ تا پروپوزال میدهیم یکی فاز اول یکی فاز دوم. در رابطه با زمان و هزینه لازم دقیق میدهیم. ۵ تا کار اصلی فاز دوم: نیازمندی ها تا حد خوبی انجام میشوند تا ۸۰ درصد و ۲۰ درصد باقی مانده هم در طول زمان و نمیتوانی همه یوزکیس ها را بدست بیاوری مثل بک اپ و ساپورت در فاز سوم مشخص میشود و تا حد خوبی scope مشخص میشود analysis و تحلیل تفضیلی خوبی انجام میشود چیستی سیستم در میاد برای design هم تا حد خوبی معماری ساخته میشود بعد در implementation هم ما architectural baseline رو میسازیم و در test همین را تست میکنیم در design کلاس ها تا حد خوبی در میان کلاس های دیتابیس ارتباط با ماشین ارتباط با شبکه و زیرساخت و طراحی تفضیلی داریم ولی عمده آن در فاز سوم هست. در همین فاز ریسک به طور کلی از بین رفته هست. یوزکیس تا حد قابل انجام و تعیین ریسک انجام میدهیم. User manual ها را به کاربر نشان میدهیم تا واکنش او را ببینیم و حساس هست تا ببینیم طراحی خوب هست یا نه. آخر این فاز تحلیل تا حد خوبی انجام شده design معماری هم تا حد خوبی انجام شده است و اون ۲ تای دیگه هم تا حد خوبی جلو رفتن. مهم ترین محصول آن executable architectural baseline با کیفیت و دقیق هست و نشان میدهیم ریسک ها تا حد زیادی حل شده اند. نمودار های تعاملی و interaction اینجا کاملا پیاده سازی میشود. یوزکیس دیاگرام sequence diagram و مشخصات غیر وظیفه ای. از ui برای تشخیص یوزکیس ها بدست میاد و نیازمندی های جدید بدست میاد. فاز سوم دیگه یوزکیس پیاده سازی میکنیم تا به سیستم هدف برسیم. ۵ تا کار اصلی در این فاز: نیازمندی ها در حد همان ۲۰ درصد. تحلیل و طراحی نهایی میشود. خروجی این پارت

محصول نرم افزاری هست که نیاز های کاربر را رفع کرده باشد به همراه UML مدل ها و تست ها و release را تا حد خوبی جلو میبریم میگیم این کدام بخش هست و توجیه اقتصادی هم چک میشود.

نکات کلیدی:

اگر یوزکیس ها یکی کلی تر هست اون رو یوزکیس تعریف کن بقیه رو زیر یوزکیس و زیر سیستم تا بتوانیم وراثت برسند تا بتوانیم ابجکت فرزندان را تا پدر جابجا کنیم. در پدر ها description و پیش شرط و پس شرط بزار و روابط را به فرزندان به ارث برسان ولی دیگر main flow را نباید توی آنها پیاده سازی کنی. و باعث میشود نمودار ساده تر میشود و ارتباطات communication کم میشود. صرفه جویی در روابط و وراثت باعث غنی تر شدن و بهتر شدن نمودار ها میشود پدر یک چارچوب تعریف میکند و در فرزندان رعایت میشود. رابطه include یک dependency هست و dependency باعث میشود تغییر در اولی باعث تغییر در دومی میشود. اگر یک یوزکیس داریم که از یوزکیس های دیگر میتواند استخراج شود یعنی اشتباه شده است چون یوزکیس هایی که ازش استخراج شده هست یوزکیس هستند چون اتمیک باید باشند include برای اشتراک گذاری هست نه شکستن. Duplication باعث ایجاد coupling میشود چون دو جا داریم و اگر یکی را تغییر بدهیم مجبوریم یکی دیگر را هم تغییر بدهیم و maintainability میاد پایین، انتشار تغییرات چیز بدی هست کلا. توصیه میشود که include یک سطحی باشد دو سطحی پیچیدگی میاورد و بیشتر از اون هم نروید. هر یوزکیس از دید کاربر اتمیک هست. extend در شرایط خاصی در یک یوزکیس میتوانیم رفتار اضافه کنیم و main flow ما بسط پیدا میکند و وقتی پیش شرط ها برقرار باشد اون extension use case اضافه میشود تحت عنوان رفتار اضافی و این یوزکیس برای اجرا شدن به اون یوزکیس پایه نیاز دارد و برعکس رابطه include هست که یوزکیس پایه برای اجرا به inclusion use case نیاز داشت. Genspeck پیچیدگی خاصی دارد فقط در صورتی که کلاس پدر را خالی کنیم. یوزکیس باید کوتاه و ساده باشد و main flow بیشتر از ۱ صفحه شد و جزئیات قلمرو جواب نباید داخل آن باشد یعنی اینکه مثلا روی این دکمه کلیک میکند غلط هست و جزئیات نباید باشد فقط چیستی باید باشد و گرنه غلط هست چگونگی نباید در یوزکیس باشد روی چیستی تمرکز کنید نباید یوزکیس وابستگی به پیاده سازی خاصی باشد به چیستی در تحلیل و نیازمندی میسازیم و یوزکیس تحلیل هست و واسه طراحی نیست. یوزکیس باید فیزیکال باشد و نباید به چیزی وابسته باشد.alternative flow ها باید در جداول دیگر باشد یوزکیس ها به صورت امری باشد یا به شکل اسم مصدر و اسم به یوزکیس میدهیم و به شکل upper camel case میدهیم. روابط برای ساده تر کردن هست اولین رابطه generalization هست یعنی اگر چند اکتور با هم همپوشانی داشتند یک اکتور اصلی تعریف میکنیم که بهش پدر هست و از پایین به بالا هست specialization یعنی یک اکتور خاص داریم و داخل آن اکتور های دیگر تعریف میشوند.

فصل سوم:

Lecture 3 – Requirements Workflow

در متدولوژی UP ۵ تا جریان کاری داریم: Requirements – Analysis – Design – Implementation – Test. جریان کاری اصلی و محوری: Requirements. در هر iteration این ۵ تا کار به ترتیب انجام میشوند ولی امروزه Implementation و Test در هم تنیده شده اند. قدرت یک زنجیر مساوی است با قدرت ضعیف ترین حلقه آن و کل متدولوژی های مهندسی نرم افزار به خاطر ضعیفی که در روش های مهندسی نیازمندی هست ضعیف می شوند.

چهار مرحله برای استخراج نیازمندی ها و مدل سازی آن ها در قالب use case به ترتیب: ۱. تعریف اولیه از مجموعه نیازمندی ها (feature) ۲. فهمیدن حوزه قرارگیری سیستم (انجام business process modeling و business domain modeling) ۳. با توجه به نتایج دو مرحله قبل، مشخص کردن نیازمندی های کارکردی در قالب use case ۴. مشخص کردن نیازمندی های غیرکارکردی. در مرحله اول، هر feature دارای وضعیت، تخمین هزینه پیاده سازی، اولویت و درجه ریسک برای پیاده سازی می باشد. Domain model یک مدل ساختاری و Business model یک مدل رفتاری است. در مرحله دوم، یک سری Domain object به دست می آوریم که در واقع کلاس های قلمرو حل مسئله هستند (که خیلی از کلاس ها به کلاس های سیستمی تبدیل نمی شوند) منتهی به شکلی که در دنیای واقعی هستند. در این مرحله package diagram یا همان subsystemهای سیستم هم تعریف می شوند. در Business model هم Activity diagram تشکیل میشود. هر پروسه در این مرحله ممکن است تبدیل به یک سرویس دهنده برای بخش های دگیر در سیستم نشود و یک کار داخلی مربوط به سیستم انجام دهد. دانشی که در این مرحله کسب میشود خیلی مهم است چون در این مرحله دید خوبی نسبت به آن چیزی که برای مشتری باید پیاده سازی شود کسب میشود. در این مرحله، اگر چندین افراد دخیل باشند بهتر است. در مرحله سوم، نیازمندی های کارکردی سیستم در قالب use case modeling

طراحی میشوند. هر use case روش استفاده از سیستم توسط یک کاربر را توصیف میکند. مجموع use caseها معادل مجموع سرویس های سیستم می باشد. هر use case model شامل use case diagram، توضیحات هر use case، glossary of times (در domain model تشکیل شده است) می باشد. Use caseها در تمام طول پروژه استفاده خواهد شد و محور انجام کار است. User کسی است که از سیستم استفاده میکند. Customer کسی است که درخواست ساخت سیستم را میدهد. Sponser کسی است که حمایت مالی میکند. در مرحله چهارم، اکثر نیازمندی های غیرکارکردی ویژگی های ناظر بر ویژگی های عملکردی هستند. بعضی از این ها ناظر بر سیستم هدف هستند مثلا reliability، maintainability، portability و غیره. بعضی نیازمندی های غیرکارکردی مثل محدودیت زمانی برای ساختن سرویس از طرف customer ناظر بر ویژگی های عملکردی نیستند.

در مرحله مشخص کردن نیازمندی های کارکردی در قالب use case، ۱. Actors ۲. Use cases ۳. Use case priorities ۴. Glossary ۵. User Interface Prototype مشخص میشوند. آکتورها کاربرهایی هستند که از سیستم ما استفاده میکنند یا سیستم هایی هستند که با سیستم ما تعامل دارند. Use caseها شامل گام های اجرایی در سیستم، عامل اجرای هر گام و نیازمندی های ویژه آن هستند. اولویت use caseها بر اساس Moscow انجام میشود. User Interface Prototype در نهایت منجر به تشکیل UI میشود و صرفا به عنوان مشتند ثبت میشود و اگر UI تغییر کند باید Prototype هم تغییر کند.

۵ فعالیت مرحله مشخص کردن نیازمندی های کارکردی در قالب use case: ۱. یافتن آکتورها و use caseها ۲. اولویت بندی use caseها ۳. اضافه کردن جزئیات و تفسیر کردن use caseها ۴. درآوردن prototypeهای user interface ۵. ساختار دادن به use case model (ایجاد روابط بین آکتورها و use caseها). در مرحله اول سیستم را از محیطش جدا میکنیم. همچنین تعیین میکنم چه کسانی یا چه چیزهایی با سیستم ارتباط خواهند داشت و چه توقعی از سیستم دارند. Glossary نیز در این مرحله تعیین میشود. هر use case باید یک نتیجه واجد ارزش برای یک آکتور داشته باشد. در مرحله دوم use caseها را بر اساس Moscow انجام میدهیم. اگر چندuse case اولویت یکسانی داشتند، آنهایی که ریسک بیشتری دارن در اولویت قرار میگیرن!! روابط بین use caseها ممکن است یک جایگاه آنها را در قانون Moscow تغییر دهد مثلا اگر یک use case، Could have باشد و use case وابسته به آن Must have باشد، آن use case دیگر Could have نخواهد بود. در مرحله سوم، یکی از مهم ترین جزئیاتی که باید اضافه شود، گام های use case می باشد. برای تعریف جزئیات از حالت شروع تا حالت پایان باید یک مسیر normal که مسیر اصلی می باشد را انتخاب کرد. مسیرهای Alternative هم هستند که از مسیر اصلی جدا میشوند و به آن بر نمی گردند. مواردی که باید در جزئیات اضافه شوند: حالت شروع و حالت پایان – زمان و چگونگی شروع و پایان use case – ترتیب مجاز گام ها – غیر از مسیرهای آلترناتیو، مسیرهای قابل اجرای غیرمجاز – اقدامات آکتورها و سیستم ها با سیستم – کاربرد آبجکت ها، ارزش ها و منابع سیستم. برای use caseهای پیچیده: از Activity diagram برای توصیف توالی فعالیت ها، از state charts برای توصیف حالت ها و تراکنش های بین آنها، از Interaction Diagram برای توصیف نحوه تعامل آکتور و سیستم ها با سیستم ما استفاده میشود. در مرحله چهارم، به دلیل اینکه بعضی سیستم های تجاری کارکردشان در UI نمایان می کنند (مثلا use caseها را در منو و ساب منو قرار میدهند) برای اینکه به کاربر ارائه دهند. در این مرحله از بازخورد کاربران نیز برای ادامه راه استفاده

میشود. ابتدا یک UI منطقی و سپس یک UI فیزیکی ساخته میشود. در مرحله پنج روابط Gens/Spec (فعالیت های رایج و مشترک بین آکتور و یا use caseها با هم)، Extend و Include تعریف میشوند.

Lecture 4 – Use Case Modeling Part 1

در یافتن آکتورها و use caseها ما ابتدا یک system boundary تعریف میکنیم یعنی یک اسکوپ فرضی (سیستم چه کاره هست و چه کاره نیست). گام بعدی یافتن آکتورها و سپس بر اساس آکتورها یافتن use caseها (و اگر alternative flow داشته باشد) میباشد. ممکن است هنگام یافتن use caseها آکتورهای جدیدی پیدا شوند. سپس همین مراحل را تا پایدار شدن آکتورها، use caseها و system boundary انجام میدهیم. Use case diagram دارای چهار عنصر میباشد: ۱. System boundary (مرز سیستم: نشان دهنده این است که ما در حال مدل کردن چه چیزی هستیم که در UML2 به آن subject میگویند.) ۲. آکتورها ۳. Use caseها (عملکردی که آکتورها از سیستم انتظار دارند) ۴. رابطه ها (بین آکتورها و use caseها). Subject را یک جعبه رسم میکنند و اسم سیستم یا ساب‌سیستم را روی آن میگذارند. آکتورها بیرون جعبه و use caseها داخل جعبه نوشته میشوند. هر چه آکتورها و use caseها پیدا شوند، subject معنی و مفهوم دقیق تری پیدا میکند. آکتور: یک نهاد یا موجودیت بیرونی که جز سیستم نیست و میخواهد از سیستم استفاده کند. در UML2، subjectها هم میتوانند آکتور باشند چون میتوانیم سیستم را به ساب‌سیستم‌ها بشکونیم و برای هر ساب‌سیستم use case بنویسیم که در این حالت ساب‌سیستم‌ها آکتور یکدیگر محسوب میشوند. آکتور زمان آکتوری است که یک کار مشخص را دوره ای انجام میدهد مثل پشتیبان گیری از اطلاعات هر روز ساعت ۵ عصر. هر آکتور شامل نام و توضیحات میباشد. نام آکتور باید معنی دار و دقیق باشد و توضیحات آن باید کوتاه باشد مثلا در حد بیان کردن مسئولیت های آکتور. Use case: توصیفی است از ترتیب های عملیات که این شامل عملیات های جایگزین و عملیات های خطا نیز می باشند که یک سیستم در تعامل با آکتورها انجام میدهد. باید توسط یک آکتور شروع شود. باید از دید کاربر نوشته شود. پیدا کردن use case جدید میتواند باعث پیدا کردن آکتور جدید بشود. Use caseها باید دارای نام کوتاه، توصیفی که به صورت عبارت فعل هستند باشند. Glossary: یک دیکشنری از واژگان کلیدی حوزه کسب و کار ما که توسط همه از جمله سهامداران نیز باید قابل فهم باشد. همچنین برای هر کلمه باید هم معنی ها و متضادها را هم در بیاوریم. اگر هر یک از واژه ها سرواژه سازی شده باشد و بخواهیم یک کلاس هم نام این واژه بسازیم اسم کلاس باید معادل واژه کامل باشد نه سرواژه سازی شده آن.