جلسه اول:

یک الگو یک راه حل موفق هست برای یک مشکل تکرار شونده در یک context خاص در هر contextیی نیست ممکن است یک راه حل در یک موضوعی درست و در یک موضوع دیگر درست نباشد مثلا در دامنه سیستم مالی درست یا در ثبت نام غلط باشد. یک الگو میتواند جز یک مرحله باشد. یک راه حل ممکن است در یک موضوع الگو باشد در یک موضوع نباشد.

جلسه دوم:

Lecture 1:

الگو: یکی از اهداف آن استفاده مجدد هست یعنی reuse و میخواهیم معماری و طراحی نرم افزار را استفاده مجدد کنیم یعنی یک کار خوب را به شکل الگو در میاوریم تا دیگران استفاده کنند. الگو هم جنبه static و هم dynamic دارد یعنی هم دنبال ساختار هستند هم دنبال همکاری object ها و اجزا در ساختار به صورت ایستا و در همکاری به شکل پویا یا dynamic تا الگو یا راه حل محقق شود. الگو یک راه حل هایی هستند برای مشکلات که در جریان ایجاد نرم افزار به صورت مکرر رخ میدهند و این راه حل ها در حل آن ها موفق هستند یک الگو یک سه تایی هست راه حلی برای یک مسئله برای یک کانتکست یا موضوع خاص چه در شی گرا یا یک مرحله از ایجاد نرم افزار یا فعالیت های مهندسی نرم افزاری یا دامنه های کاربردی برنامه ها یعنی انواع سیستم ها همه اینها میتوانند کانتکست ما باشند البته paradigm ها یا دیدگاه ها مثل شی گرا به برنامه و سیستم ما هم جز همین کانتکست ما هستند. paradigm همان دید و چگونگی تلقی از سیستم ها و رویکرد در مورد سیستم ها هست. یعنی اون راه حل در اون کانتکست خاص تبدیل به الگو میشود و جزئیات خاص مربوط به اون موقعیت را از آن حذف میکنند و یک لایه abstraction میزنند تا قابلیت استفاده مجدد بالا رود. راه حل را باید از دو دید تعریف کنیم یک structural یعنی سیستم ما چه کلاس هایی دارند و چه روابطی دارند و دو این راه حل ما به چه شکل در زمان اجرا رفتار میکند و اجزا چگونه پیام میفرستد تا راه حل محقق شود و این همکاری پویا و dynamic هست ولی ساختار ایستا هست و از هر دو جنبه باید مدل و بیان شود. هر الگو یک کلاس دیاگرام دارد که نشان میدهد چه کلاس هایی به حالت before اضافه شدند و تعامل دارند در زمان اجرا ابجکت های آنها تا راه حل موفق شود برای ساختار از کلاس دیاگرام و برای تعامل بین ابجکت ها از sequence diagram استفاده میکنیم. راه حل هم باید structraul باشد یعنی راه حل از چه اجزایی و چه کلاس های تشکیل شده است و این کلاس ها با هم چه ارتباطی دارند و بعد در زمان اجرا به شکل رفتار را بروز میدهد یعنی چطوری به هم پیغام میفرستند تا راه حل محقق شود و این همکاری جنبه dynamic دارد منظور رفتاری هست structure یک چیز static هست. برای هر الگو هم ساختار و هم رفتار را مدل کنیم. هر الگو یک کلاس دیاگرام دارد که نشان میدهد الگو چطوری محقق میشود یعنی چه کلاس هایی به حالت قبل اضافه میشوند و این کلاس ها چه روابطی با همدیگر دارند و ۲ چگونه

اشیا آن ها در زمان اجرا با هم تعامل میکنند تا این الگو محقق شود برای جنبه ایستا از کلاس دیاگرام استفاده میکنیم و برای جنبه رفتاری از sequence diagram استفاده میکنیم یا برای همکاری ها. الگو یک ۳ تایی هست: solution, problem and context یعنی یک راه حل برای یک مسئله در یک زمینه خاص. کارت CRC برای استخراج کلاس ها در سیستم ها به کار میروند. Idiom یعنی یک سری راه حل خاص در یک زبان برنامه نویسی و نه به صورت کلی.

الگو طراحی نام گذاری میکند انتزاعی میکند و شناسایی میکند جنبه های کلیدی یک ساختار رایج طراحی برای ساختن یک طراحی شی گرا استفاده مجدد انجام میدهد. در طراحی وقتی به قلمرو مسئله سیستم را تعریف میکنیم راه حلی را برای حل مشکل پیدا میکنیم که این به کرات مورد استفاده قرار میگیرد بعد از شناسایی کردن باید نام بگذاری و انتزاعی بکنی یعنی جزئیات رو بزنیم که یکم عام تر بشود نسبت به یک موضوع خاص نباشد جزئیات زائد رو بگذاریم کنار. اگر نسبت به یک حوزه خاص حذف شود استفاده مجدد آن بالاتر میرود. هر ۲ تا را باید مدل کنیم هم ساختار هم رفتار. یکی از مزیت های الگو های طراحی این است که جنبه های non functional را تقویت کنیم و تلاش میکنیم نیاز های غیر وظیفه ای را محقق کنیم و مشکلاتش بر طرف شود در خیلی از موارد نیازمندی غیر وظیفه ای هست نه نیازمندی وظیفه ای. برای استفاده از الگو ما همیشه یک چیز را میدهیم و یک چیز را میگیریم استفاده از الگو باعث افزایش استفاده مجدد یا reusability میشود ولی هدف اون نیست بلکه هدف مشکلات flexibility هستش یعنی تغییر پذیری کد آمده پایین یا coupling رفته بالا یا به اصطلاح کد solid شده است اگر این ها بالاتر برود reusability و flexibility کد پایین میاد. اگر coupling بالاتر رود maintainability پایین می آید و اگر نرم افزاری maintainability پایینی داشته باشد مرده است و در هر سیستمی باید باشد. Reliability and robustness را میتوان تا حدودی گذشت یا قربانی کرد ولی اصلا maintainability را نمیتوان قربانی کرد تمام تاریخچه مهندسی نرم افزار و زبان های برنامه سازی برای همین افزایش maintainability هست همه در جهت بالا تر بردن cohesion و کاهش coupling برای افزایش تغییر پذیری تا باز هم maintainability محقق شود یا برای افزایش capsulation. Maintainability یک نیاز غیر وظیفه ای اساسی هست و هیچ گونه نمیتوان این را قربانی کرد. ۲ معیار برای مهندسی نرم افزار داریم cohesion, coupling برای بالا بردن maintainability. Coupling بالا یعنی چی یک جا را تغییر میدهیم ولی به خاطر اون تغییر کل قسمت ها تغییر میابد. Maintenance ۴ نوع دارد ۱. Corrective یعنی یک جایی از کد خراب میشود تصحیح میکنیم. ۲. Perfective یعنی وقتی feature جدید اضافه میشود ما تکمیل میکنیم کد را ۳. Adaptive یعنی تطبیقی یعنی تغییری در زیر ساخت یا پلتفرم ایجاد میشود و ما باید نرم افزار را تطبیق بدهیم. ۴. Preventive یعنی برای اجتناب از بروز یک مشکل در آینده داریم تغییرات میدهیم. معمولا در توسعه امروزی مشغول دومی هستیم یعنی هر تکه

ای که میسازیم باید به تکه های قبلی اضافه کنیم و خود این اضافه کردن نیازمند تغییر در بخش های قبلی و اضافه کردن آن هست. وقتی یک ماژول به دلایل بی ربط مثلا عوض شدن UI عوض شد اینم عوض شود به اصطلاح cohesive نیست یعنی تک کاره نیست نقض این باعث بروز divert change میشود. تحلیل نرم افزاری یعنی مشخص کردن چیستی سیستم که ۲ مرحله دارد preminil analysis و detailed analysis. در اولی یک مقدار اطلاعات برای امکان پذیری سیستم را میسنجیم در دومی اطلاعات کلی و کامل راجب اون سیستم را بدست میاوریم. در تحلیل فقط دنبال چیستی سیستم هستیم دنبال چگونگی پیاده سازی نیستیم این برای طراحی هست طراحی هم ۲ بخش دارد معماری و تفضیلی در معماری ساختار سطح بالا و component های سطح بالا و تکنولوژی های قلمرو راه حل در میان یعنی UI چه طوری باشه پروتکول های ارتباطی یا ارتباط با زیر ساخت و ... اینها در طراحی معماری شناخته میشوند. در طراحی تفضیلی جزئیات دقیق اجزای سیستم و مشخصات و عناصر تشکیل دهنده سیستم را بدست میاوریم یعنی همان کلاس های سیستم به طور به شدت کامل و وارد جزئیات ریز دانه میشویم و میدهیم به برنامه نویس. یک سری الگو برای معماری ها داریم به اینها الگو های معماری گفته میشود. یک سری برای طراحی تفضیلی که خیلی ریز دانه هستند الگو های طراحی نام دارند.

الگو های کد: roles played = state و broadcast = observer سمت راستی ها اسم دیگر سمت چپی در ها در گنگ چهار نفره هستند. بقیه الگو های کد خیلی ریز دانه هستند و نمیتوانند جز الگو های طراحی بیایند راه حل ریز دانه برای مشکلات ریز دانه هستند. item description: میخوایم یک فقره اطلاعاتی را توصیف کنیم مثلا فرض کن یک object به نام ایتم داریم و یک ابجکت دیگر میخواهیم برای توصیف آن ایتم تا توصیف آن ایتم را از خودش جدا کنیم یعنی از داخل اون ابجکت یک کلاس دیگر را بیاوریم بیرون یعنی این یک ابجکت نیست ۲ تا باید باشد وقتی مقادیر تکرار شوند بین instance های آیتم پیدا میشود یعنی بعضی از attribute ها برای هر instance مقدار خاص و برای بعضی دیگر این مقادیر تکرار میشوند. یعنی انگار همون افزونگی داخل دیتابیس هست. مثلا فرض کن یک کلاس داریم به اسم car و attribute های بعضی ها خیلی خاص هستند مثل شماره پلاک و برای هر ابجکت یک معنی خاص دارند ولی در کنار یک سری چیز هایی دیگری داریم که مقادیر تکرار شونده دارند مثلا اینکه مدل از چه نوعی هست یا مثلا تعداد سرنشین آن چه قدر هست. برای این car باید یک مدل تعریف بکنیم که هر ابجکتی از car با یک ابجکت از مدل در ارتباط هست مدل همون مدل ماشین هست چون ممکن هست تکراری باشد و attribute های مدل داخل همان کلاس مدل هستند و دیگر جز car نیستند. وقتی جدا میکنیم افزونگی از بین میرود ولی یک ابجکت شد دو ابجکت یا ۲ کلاس و یک رابطه ایجاد شده و باید پیمایش کنیم از ماشین به مدل. هر موقع attribute مقادیرش تکرار شونده بود یعنی به بیش از یک ابجکت اطلاق شد یعنی احتمالا اون ابجکت داخل یک ابجکت دیگر هست و به دو ابجکت نیاز داریم. در نتیجه این کار cohesion بالا میرود چون کلاس مرکب ترکیب به

یک کلاس شده است و تک مقصود بودن کلاس بالاتر میرود ولی یک حدی از coupling هم بالاتر میاد بخاطر اون رابطه. در مورد الگو ها یک چیزی را بدست میاورند و یک چیزی را قربانی میکنند. معمولا برای افزایش cohesion و کاهش coupling هست ولی ممکن هست یک چیزی از بین برود. مثل همینجا cohesion رفته بالا و coupling هم کنارش رفته بالا. در بیشتر الگو ها چیزی که از دست میرود efficiency هستش چون عناصر درگیر تعداد زیادتر میشود یعنی پیچیدگی بالاتر و تعامل بالاتر و این یعنی کندی. امروزه حدی از efficieny که قربانی میشود را میپذیریم چون مشکل جدی ایجاد نمیشود. ولی در بعضی دیگر از انواع الگو ها مثل decorator، سیستم ممکن است به شدت کند بشود و یک سری cascade از تعاملات زائد به وجود میاد یا observer هم همین مشکل وجود دارد همه الگو ها یک سری تبعات های منفی هم دارند. Mediator یک الگو رفتاری هست رایج ولی خطرناک هست واسه وقتی هست که بین ابجکت ها تعاملات زیاد هست و coupling بالا هست برای حل این مشکل یک میانجی میگذاریم تا ارتباط مستقیم بین ابجکت های دیگر از بین برود و فقط mediator را میبینند و دیگر همدیگر را نمیبینند و coupling فقط یک طرف میشود و چند گانه نمیشود و یک تغییر به همه منتقل نمیشود و باعث کاهش وابستگی بین همدیگر ولی باعث گلوگاه یا god class شدن سیستم بشود و single pointer failure بشود یعنی چی یعنی اگر operation های بعدی را خواستیم تعریف کنیم به جایی اینکه بشکنیم در ابجکت ها یک راست در این میانجی قرار میدهیم و این میانجی رشد میکند فقط عملیات ها به این اضافه میشود به جای شکسته شدن در ابجکت های همکار و میانجی هست که داده ها را از ابجکت میگیرد و کار را خودش انجام میدهد. god class یعنی کلاسی که پر از رفتار هست ولی داده کنار آن نیست و به شدت cohesion آن پایین هست و کار های بی ربط در آن هست و دستکاری آن خیلی سخت هست. به ۲ دلیل سیستم های شی گرا میمیرند یکی cohesion پایین بخاطر ایجاد god class پیچیدگی بالا و دستکاری سخت و دومی ایجاد message chain یعنی زنجیره ای از ابجکت ها که به این شکل هست که ابجکت سر زنجیره یک درخواستی از ابجکت بعدی میکند ابجکت بعدی به جای اینکه به بعدی تحویل بدهد میاد بعدی را در اختیار ابجکت سر زنجیره میگذارد. یعنی هعی ابجکت های داخل زنجیره دیگری را معرفی میکنند و ابجکت سر به همه ابجکت ها دید پیدا میکنند و این باعث از بین رفتن maintainability میشود و باعث افزایش coupling میشود. در facade یک ابجکت میشیند روی یک زیر سیستم همین مشکل اونجا هم همین هست یعنی مشتری های اون زیر سیستم میومدن با کلاس های اون کار میکردن coupling بالا بود برای کاهش آن اومدیم یک ابجکت قرار دادیم. شی گرایی یعنی عملیات نزدیک داده قرار بگیرد. Façade یک wrapper هست همیشه یک چیزی از دست میرود ولی باید ببینیم می ارز یا نه.

جلسه سوم:

اگر کلاس ها داخل نمودار دو تا خط بیرون نداشتن یعنی abstract هستند و instance ندارند. رابطه associationرابطه ای بین instance ها هست. رابطه construction یعنی به کلاس میگی یک instance از خودت بساز. Association بین قاب بیرونی هست و ابجکت ها هست. item description بعد از استخراج یک کلاس از شکم کلاس دیگر رابطه association برقرار میکند. Time association: اگر یک شرایطی بود که یک سری داده روی یک کلاس درگیر قرار نمیگرفت بلکه روی یک رابطه قرار میگرفت اونجا باید از این استفاده کنی و احتمالا یک کلاس داخل اون کلاس خوابیده و باید استخراج کنی به اصطلاح باید reify کنی که حاصل رابطه association بوده و حالا دیگر یک کلاس کامل هست و داخل این کلاس باید تاریخ این جدا شدن ثبت شود. یک مثال مثلا نگاه کن پیگیری سفارش نه مربوط به محصول هست نه مربوط به کاربر چون یک کاربر چندین محصول دارد و یک محصول چندین کاربر این باید روی سفارش بشیند که مستتر هست در رابطه و باید کلاس مربوطه را reify کنیم کلاس وسطی ارتباط دهنده طرفین هست. و این سفارش کلاس اصلی سیستم هست و این ارتباط دهنده طرفین میشود.

Event logging: این برای سیستم های real time هستند و نیاز دارند برای پردازش event ها بیرونی سرو کار دارند مورد استفاده قرار میگیرد. یعنی یک سنسوری یک چیزی را سنس میکند و بعد از آن باید آن را پاسخ بدهیم. و این سیستم monitor میکند چندین سنسور را. لزوما خودش پاسخ سنسور را نمیدهد ممکن است بدهد به یک ابجکت دیگر تا اون کار را انجام بدهد. event remembered یک سری attribute دارد که مقادیرش را بر اساس اون event رخ داده پر میکنیم و به ازای هر رویداد یک ابجکت از این کلاس میسازیم. یا مثلا باید یک واقعه رو از طریقlog ثبت بکنیم. هر موقع خواستیم یک رویداد را ثبت کنیم باید به این الگو فکر کنیم.

الگو بعدی roles played: این الگو state هست البته قبل تر از اون هست. این ابجکت های ما در طول زمان رفتار های متفاوت از خودش بروز میدهد. بر اساس نقشی که دارد رفتار میگیرد مثلا دانشجو یک رفتار وابسته به حالت دارد یعنی ابجکت رفتار وابسته به حالت دارد و برحسب اینکه در چه حالتی هستند رفتار متناسب را از خود بروز میدهند. به جای اینکه ابجکت کشته بشود و اطلاعات آن از این ابجکت به ابجکت دیگری برسد. Role ها حالت های اون ابجکت هستند و بر اساس اون رفتار متفاوت دارند نقش باعث تغییر رفتار میشود. مثلا دانشجو میتونه مشروط باشد یا نباشد یا ... . یعنی از جنسیت شروع میکنیم هعی بر اساس حالات درخت طور میایم پایین و تقسیم میکنیم و ترکیب های مختلف میسازیم و همه ترکیبات در برگ ها هستند و کلاس ها در برگ ها هستند. به ازای هر سطح به صورت افزایشی کلاس اضافه میشود و ترکیبات جدید میاد و زیاد میشود و این ساختار کلاسی درختی برای ساختن حالات خیلی مناسب نیست حتی اگر حالت عوض شود اون ابجکت باید حذف شود ابجکت جدید ایجاد شود و موارد آن به ابجکت جدید برسند. وقتی مبنای زیر کلاس

یک چیز متغیر باشد دردسر میشود چون هعی باید کلاس زیاد ایجاد شود و چون تغییر حالت امکان پذیر هست و نباید مبنای زیر کلاس قرار بدهیم مثلا مشروط شدن چون مثلا دانشجو مشروط بشه باید اون ابجکت کشته شود و ... ، ولی این الگو این مشکل را حل میکند به این صورت که حالت های ابجکت ها را به صورت ابجکت حالت در شکم خودش تعریف میکنیم یعنی علاوه بر اون attribute ها مثل شماره دانشجویی اون حالاتی هم که دارد به عنوان یک ابجکت در شکم خود دارد. بعد اگر حالت عوض شد به ابجکت داخلی مربوطه میدهد تا اون انجام بدهد پس وقتی یک ابجکت داشتی که خودش هویت مستقل داشت ولی بر اساس حالات و رفتاری که دارد مخصوصا رفتار یعنی operation ها یکسان نیست و نسبت به راه حل توارث راه حل بهتری هست. استفاده بد از وراثت به هدف استفاده مجدد بد هست و باید با هدف delegation استفاده کنیم. Gen/spec همان رابطه is a هست هر gen/spec یک وراثت هست. خالی کردن رفتار یا operation در یک زیر کلاس یک کار بد هست و به اسم refuse bequest یا میراث مردود شناخته میشود. override به معنی خالی کردن بی معنی هست و is a نقض میشود و یک زیر کلاس نمیتواند رفتاری که از بالا میرسد را محدود تر بکند و is a نقض میشود و یک میراث مردود هست امروز اگر extend کنی یعنی زیر کلاس تعریف کنی و بهش رفتار جدید اضافه کنی زیر کلاس برای extend کردن هست. نیم دایره در نمودار gen/spec و مثلث رابطه association هست. یعنی به جای اینکه کلاس جدید تعریف کنیم داخل شکم آن ابجکت های role هست. وقتی یک چیز متغیر مبنای زیر کلاس شدن شد حواست باشه که دچار این مشکل نشوی و الگو را در خاطر داشته باشی. دقت کن در این الگو ترکیب انواع حالات مد نظر نیست یکی از اینها در هر لحظه موضوعیت دارد و زمان شروع و پایان حالات به همدیگر میچسبد. به این شکل ابجکت اصلی سبک میشود و در زمان اجرا یک ابجکت داخل میگذاریم و به آن کار را محول میکنیم. و اون کار جدید delegate میشود به ابجکت داخل شکم و این کاملا انعطاف پذیر هست و ابجکت از بین نمیرود. CRP میگوید هر جا تونستی وراثت رو بنداز دور و از delegate استفاده کن زیرا انعطاف بیشتری دارد و این نسبت به اون مقدم تر هست. CRP component reuse principle که دقت کن خود delegation داخل خودش وراثت را هم دارد این راه حل بهتری هست. CRP یک اصل مبنای راجب گنگ چهار نفره هست.

State over a collection: یک collection داریم یعنی یک مجموعه عناصر و یک حالت داریم حالت شامل ۲ چیز هست یک مقادیر attribute ها و ابجکت ها دیگری که میشناسد. یک ابجکت collection داریم که حالت خودش را میشناسد یعنی مقادیر attribute ها و بعد اون ارتباط اون attribute ها با ابجکت های دیگر دارد. هر کدام از اعضا این کالکشن میتواند حالت خاص خودش را داشته باشد. این الگو وقتی کاربرد دارد که یک رابطه کل به جز در قلمرو مسئله یا در قلمرو جواب یک رابطه کل به جز داریم و برخی از attribute ها مربوط به کل هستند ولی در جز ها قرار داده شدند.

Behavior over a collection: این هم زوج همین الگو قبلی هست منتها به جای حالت یا داده چون حالت بیشتر به attribute تمایل دارد رفتاری که برای کل هست درون جز ها باقی مانده هست. و رفتار های اجزا باید در خودش و رفتار کل باید در کل بماند. حتی برعکس هم غلط هست رفتار جز نباید در رفتار کل باشد چون god class میشود و اجزا فقط داده دارند. Data class یک چیز بد هست چون فقط داده هست و رفتار نیست. whole-part همان رابطه aggregation هست و ۳ وضعیت دارد که میگیم رابطه کل به جز دارند ۱. وقتی ابجکت های یک سمت تشکیل دهنده ابجکت های سمت دیگر باشند که بهش assembly گفته میشود. ۲. Containment یا ظرف و مظروف یعنی یک طرف ظرف هست و اون طرف مضروف، که درون ظرف قرار میگرد این از اولی سبک تر هست چرا؟ چونکه تو اولی کل به جز بدون هم معنی ندارند هواپیما بدون موتور معنی ندارد ولی در اینجا ظرف میتواند باشد حالا پر باشد یا نباشد البته این هم رابطه کل به جز هست. ۳. Membership یعنی یک طرف عضو اون یکی طرف دیگر هست بین تیم و اعضای تیم که این هم یک رابطه کل به جز هست. به کل این ۳ تا whole-part یا aggregation گفته میشود و این یک رابطه قوی تری هست از association چون تو این رابطه instance با همدیگر متناظر هستند و به هم دید دارند مثلا از طریق attribute دید مانا دارند ولی این ضعیف تر از قبلی هست اول association میکشیم بعد تبدیل میکنیم به aggregation بعضی ها را چون قوی تر هست و این دید یک طرفه دارد یعنی فقط از کل به اجزا هست و این پیاده سازی آسان تر هست. دقت کن اون رفتار ها اگر در کل مجموعه صدق میکنند باید در طرف کل گذاشت. دقت کن رفتار های اجزا باید تا جایی که میتوانند کار خود را انجام بدهند. دقت کن کل رابطه را هماهنگ میکند روی اجزا اون غالب هست. catalog یک ابجکتی هست که داخلش ابجکت های دیگر قرار دارد مثلا بین سرچ کردن یک catalog داریم که به جای اینکه اجزا که کتاب ها مثلا هستند سرچ کنند اون کل که catalog هست بیاد سرچ کند. در زمان اجرا فقط ابجکت ها هستند که باعث تحقق میشوند و کار را انجام میدهند نه کلاس ها کلاس ها فقط قالب هستند و کننده کار نیست ابجکت کننده کار هست و اشتباه در شی گرا هست. الگوی single tone میگوید ما میتوانیم یک کلاس داشته باشیم با یک تک instance و این اولویت دارد نسبت به static ها چون ایستا بشود دیگر extend کردن آن سخت هست. اگر یک رفتاری داشته که روی مجموعه دروس یا روی مجموعه عناصر تاثیر داشته باشد و رابطه کل نداشتیم یک catalog تعریف میکنیم که همین هست چون روی همه عناصر تاثیر دارد. بعضی از انواع aggregation خیلی قوی هستند و اسم خاص دارند. رابطه قوی تر را بهش composition گفته میشود البته یکی از انواع aggregation هست که خیلی قوی تر هست خوب حالا این رابطه دو ویژگی خاص دارد ۱. ممکن است lifetime داشته باشیم یعنی ممکن هست کل بعد یک مدتی از بین برود پس باید اجزا هم از بین برود و عمر اجزا به عمر کل وابسته هست ۲. Sharing نداریم یعنی جز فقط و فقط جز یک کل هست نه چند تا کل حالا اگر کل از بین رفت باید اون اجزا به یک کل دیگر برسند. Composition پیاده

سازی آسانی دارد چون اجزا در شکم کل هستند. خصوصیت سوم این هست که مستقیم به اجزا نمیتوانیم دید داشته باشیم و باید از کل نگاه بکنیم. اگر ۲ تا شرط اول را داشتیم میگوییم composition داریم. در این رابطه کل مالک مطلق این اجزا هست. در پیاده سازی هم خاصیت سوم ضرورتی ندارد ولی به صورت encapsulation را معرفی میکنند یعنی جز داخل کل قرار میگیرد و coupling را کمتر میکند و از بیرون دید ندارند.

الگو broadcast: یک نوع خاصی از observer هست که برای انتشار تغییرات بین زیر سیستم یا اجزا درشت دانه قرار میگیرد. ستون های سیستم ۳ تا ستون دارد UI BUSINESS LOGIC و data. این انتشار تغییرات دارد بین این ۳ تا ستون یا قسمت درشت دانه سیستم. یک عنصری داریم که بهش میگیم subject و یک تعداد ابجکت داریم که به تغییرات این حساس هستند و اگر تغییر کرد باید به اطلاع بدهند با کمترین coupling موجود باید این کار را انجام بدهیم. بهترین روش این هست که در سمت حساس به تغییرات بگیم تغییر انجام شده بدون اینکه بگیم دقیقا چی انجام شده و در سمت subject هم نمیدانیم اینها چی هستند و هیچ دانشی نداریم و فقط میدانیم این ابجکت ها حساس به تغییر هستند و فقط لینک داریم که اطلاع بدهیم حتی وقتی تغییر میشود هم نمیدانیم الان حساس هستند یا نه فقط یک آدرس داریم و فقط میگیم اپدیت ساده ترین روش پیاده سازی این هست که از سمت subject یک لیستی داریم از اینها که حساس به تغییر هستند و به اونها observer گفته میشود اینها میتوانند از کلاس های بسیار متنوعی باشند در یک چیز مشترک هستند و فقط interface سابجکت را پیاده سازی میکنند. Observer ها خودشان را پیش سابجکت subscribe میکنند تا اون لیست تشکیل شود در واقع یک متد هست. سابجکت یک عملیاتی به نام notify دارد و لیست را پیمایش میکند و فقط میگوید اپدیت اونها اگر حساس به تغییر باشند میاد حالت رو میگیرند اگر مهم باشد تغییر میدهد و نمیخواهیم هیچ اطلاعی بین طرفین باشد چون باعث افزایش coupling میشود. دقت کن متد unsubscribe داریم که اگر دیگر به تغییر حساس نبودند این فعال میشود. هر چی اطلاع کمتر coupling کمتر. انتشار تغییرات برای طراحی هست عمدتا البته میتواند برای تحلیل هم باشد ولی لزوما نیست و تو معماری ۳ ستون که قبلا گفتیم هم هست یعنی انتشار تغییرات بین اینها تغییر میکند. دقت کن بعد از اطلاع دادن باید get state کنند تا ببینند نسبت به اون تغییر حساس هستند یا نه state یعنی مجموعه مقادیر attribute ها و ابجکت هایی که اون ابجکت اصلی میشناسد این الگو برای ارتباط تعاملی بین عناصر درشتانه سیستم و به شکل کاهش coupling استفاده میشود coupling هیچ موقع صفر نمیشود هر موقع ارتباط هست اون هم هست. facade یک interface ساده ایجاد میکند و سایرین فقط یک واسط ساده میبینند و از اون استفاده میکنند و از دیدن داخل اون زیر سیستم های پیچیده دیگر نگاهی ندارند. کل به جز دیدی برو اون state behavior collection و همچنین روی چند تا چیز هست اونایی که روی همه ایتم ها هست برو بالا اونایی که جزئی هست پایین. دقت کن در حالت قبلی همه اطلاعات محصول تو قسمت تراکنش فروش بود و این اطلاعات هعی تکرار میشد ولی یک item description زدیم این مشکل را

حل کردیم. در تعامل بین کارمند و مشتری تراکنش یا فروش ظهور میکند ولی هنوز کالا ظهور نکرده بعد، بعد از این مرحله بقیه چیز ها نمود پیدا میکند.

Lecture 2:

الگو های گنگ چهار نفره:

دسته اول:

Creational:

اصول کلی: تاکید زیادی روی انعطاف پذیری و استفاده مجدد با استفاده از کاهش coupling داریم. تاکید روی یک سری قوانین: ۱. کلاس های concreate رو با هم در ارتباط مستقیم قرار نده وابستگی را با واسط interface بگذار نه به صورت مستقیم. ارتباط مستقیم در تحلیل هست ولی در طراحی نباید مستقیم باشد و باید بهش فکر کنیم برای کاهش وابستگی به این DIP هم گفته میشود که یک از اصول شی گرایی هست. dependency inversion principle یعنی به جای وابستگی در سطح concreate در سطح بالا وابستگی تعریف میکنیم یعنی از فوق کلاس ها وابستگی داریم و کلاس های ریز به هیچ کدام از جزئیات همدیگر با خبر نیستند و فقط با واسط کار میکنند. وابستگی نباید به پیاده سازی باشد. روابط غیر مستقیم میشود و تغییرات باعث تغییر در جای دیگر نمیشود مگر اینکه واسط عوض بشود. ۲. Composition را به inheritance ترجیح بدهید یعنی چی؟ یعنی delegation به جای ارث بری یعنی بگو یک شخص داریم که شخص یک کار دارد حالا اون کار میتواند کارمند یا هر چی باشد این رابطه مستقیم ارث بری نباشد. ۳. چیزی که تغییر پذیر هست را encapsulate کن تا تغییر آن به جای دیگه نرسد و کاهش وابستگی دارد.

جلسه پنجم:

DIP باعث بروز OCP میشود یعنی بتوانیم سیستم را بسط بدهیم و گسترش بدهیم بدون اینکه بقیه قسمت ها عوض بشود. تغییر سیستم ها را میکشد. CRP یا composite reuse principle یعنی delegation به ارث بری ارجحیت دارد یعنی به جای اینکه اون رفتار را به ارث بگیری بیا و سرویس بگیر از یکی دیگه چون ساختار توارثی به شدت سلب هستند. حالت عوض بشود فقط ابجکت داخلی عوض میشود و رفتار عوض میشود و ابجکت اصلی مثلا دانشجو از تغییر رفتار با خبر نمیشود بخاطر CRP هست. فرق multiple classification با single classification: classification یعنی تعلق یک ابجکت به یک کلاس اولی یعنی یک ابجکت همزمان به چند کلاس ابجکت باشد دومی هم مشخص هست. dynamic classification with static: دومی یعنی یک ابجکت بتواند کلاسش را عوض کند یعنی از یک کلاس تولید شده ولی کلا کلاسش و رابطه اش را قطع کند اولی

هم مشخص هست ایستا هست تا آخر با همان کلاس اولی میماند. کتاب معرفی شده آن را بخوانید. یک بخشی از سلب شدن ساختار توارثی بخاطر همین ۲ تا مفهوم قبلی هست چون ایستا و مفرد هست کلاس بندی ما. ساختار توارثی برای استفاده مجدد غلط هست باعث نقض is a میشود مثال اسب و عنکبوت و باعث میراث مردود میشود قانون جابجایی list of میگوید باید ابجکت زیر کلاس را باید بتوانی جای ابجکت کلاس فوق بزاری. از ارث بری برای چند ریختی یا generalization استفاده میکنیم و فوق کلاس آن abstract هست چون معمولا برای ساختن ابجکت نیست و نیازی ندارد پس abstract تعریف میشود. ۳ دسته الگو داریم از این ۲۳ تا اولی creational یعنی ابجکت بسازیم از کلاس بعد پیکربندی کنیم هدف ایجاد ابجکت هست و اگر اشتباه انجام شود وابستگی زیاد میشود. structural یعنی اینکه جدا کردن واسط ها و پیاده سازی ها هست به طوری که هر کدام به طور جدا رشد کنند Behavioral. Class امروزه وقتی میگوییم بسط میدهیم یا extend یعنی ابجکت میسازیم و گسترش میدهیم. scope یعنی الگو در زمان کامپایل محقق میشود یا بر اساس ساختار توارثی و gen/spec محقق میشود. object scope اینجا دیگر توارثی نداریم و راه حل منعطف که delegation هست استفاده میشود و اینها در زمان اجرا محقق میشوند و باید خودمان رابطه را برقرار کنیم.

Factory method: با استفاده از توارث محقق میشود و خودش یک الگوی ریز دانه داخل یک الگوی درشت دانه هست یعنی در الگو های دیگر میشود استفاده کرد و class scope هست در زمان کامپایل محقق میشود. میخواهیم یک فوق کلاس داشته باشیم که اون یک واسطی داشته باشد که برای ایجاد ابجکت ها باشد ولی خود ایجاد توسط زیر کلاس ها صورت بگیرد. فوق کلاس باید abstract باشد و polymorphic هست و ایجاد رو polymorphic کردیم و به این متد factory گفته میشود و این کار به زیر کلاس ها محول میشود و اونها factory را پیاده سازی میکنند. بین operation و متد فرق دارد specification operation هست و پیاده سازی ندارد. متد پیاده سازی عملیات هست و متد بدنه دارد یک operation میتواند با متد های مختلف پیاده سازی شود. کلا تو فوق یک hook هست که زیر کلاس ها اون رو ارث بری میکنند. Operation های italic یا hook هستند یعنی یک رفتار پیشفرض درون آنها هست یا خالی هستند و پیاده سازی آن به زیر کلاس ها داده میشود. italic دیدی یعنی abstract. توارث با مثلث و لوزی میشود aggregation و خط چین میشود instanciation و یک جور وابستگی هست. روابط به فرزندان به ارث میرسد. یعنی روابط داخل کلاس فرزندان آنها هم این روابط را دارند. مثال: ما چند نوع مثلا doc داریم ولی لازم نیست متناسب با هر کدام یک الگوریتم تعریف کنیم بلکه صرفا رفتار کلی را داریم حالا اگر چیز دیگری هست به پایینی ها میدهیم تا اونها بر اساس نو خاص doc آن را پیاده سازی کنند و داخل کلاس بالایی نمیگذاریم. Framework چهارچوب های قابل استفاده مجدد چه تو طراحی و چه تو تحلیل هستند که به صورت عمومی سازی شده یک سری پروژه های خاص بودند.

پس این الگو واسه وقتی هست که یک کلاس نمیتواند پیشبینی کند از چه کلاس های دیگر باید ابجکت بگیرد حتی اگر هم پیشبینی کند باز معقول نیست که سنگین شود پس به زیر کلاس های مختلف میسپریم و کار را polymorphic کردیم و هر زیر کلاس متخصص doc خودش هست در حالت استفاده از helper ها هم استفاده میشود مثلا برای تغییرات یک شکل که در کلاس متخصص سرویس دهنده گذاشتیم و میداند وقتی سرویس خواست کدام ابجکت را باید بگیرد. سرویس گیرنده میداند کدام سرویس دهنده مربوط به او هست و با دیگران ارتباط ندارد. در نمودار بعدی نقض DIP داریم چرا چون concreate ها با هم مستقیم در ارتباط هستند و بعد تازه به ازای هر تولید کننده یک کالا جدید هم تولید میشود و چون وابستگی ها در پایین هستند مدام دچار تغییر میشوند یعنی به ازای تولید کننده همه تغییرات میشوند یعنی از سمت factory تغییر شد در helper هم تغییر میشود و DIP نقض میشود و باعث افزایش وابستگی هست و ساختار توارث موازی چون با هم رشد میکنند مطلوب نیستند و الگو بعدی این مشکلات را رفع میکند. نکته آخر راجب factory و اون مثال: نگاه کن اگر این الگو نبود باید خود برنامه اپلیکیشن کلا کلاس ها و زیر کلاس های نوع های مختلف doc رو میشناخت و در زمان مناسب از آنها سرویس میگرفت ولی الان نیازی به این حرکت نیست.

جلسه ششم:

برای دید مانا نیاز به association یا aggregation نیاز داریم ولی شاید هم لازم نباشد و با توارث کافی هست در این الگو factory و دید مانا ندارد و غیر مانا دارد. اگر هزینه instanceciation ساده باشد زیاد استفاده از این الگو صرف ندارد. از داخل DIP نیست ولی از بیرون هست زیرا با creator یک کلاس انتزاعی هست و instance هر کدام از این زیر کلاس به کلاینت میتواند داده شود و OCP برقرار هست از دید کلاینت های بیرونی چون زیر کلاس ها میتوانند رشد کنند ولی اگر creator رو concreate میکردی حتی اگر ساده بود دیگر این برقرار نبود و DIP هم برقرار نیست. اون factory باعث میشود کلاینت فقط با واسط کار میکند و زیر کلاس ها را نمیشناسد و نمیتواند ابجکت بگیرد پس نیاز دارد به یک factory که اون بتواند از این زیر کلاس ها ابجکت بگیرد چون اون انها را میشناسد. خیلی از الگو ها به یک ثالث نیاز دارند و اونها هستند که الگو را محقق میکنند چون روابط با ابجکت ها باید با هم شروع به کار کنند در اینجا DIP برقرار هست چون کلاینت هیچ تغییری را نمیفهمد و میتوانی زیر کلاس تعریف کنی چون فقط واسط میشناسد و اصلا زیر کلاس نمیشناسد. اونی که abstract باشد میشود factory method دقت کن ساختار توارث موازی شکل گرفته و concreate داریم و داخل ساختار DIP نقض میشود. دقت کن به ازای تغییرات جدید کلاینت فقط نباید تغییر کند. در instanciation دید مانا نیست از طریق attribute میتوانیم دید مانا داشته باشیم. موارد استفاده: میخواهیم در بخشی از سیستم که فقط سرویس گیری هست استقلال داشته باشیم از اینکه محصولات چطوری ساخته میشوند و داخل آنها چی هست و ساختار داخل آنها چی هست و کلاینت نباید به اینها وابسته باشد یعنی مورد استفاده بودن از ساخته شدن محصولات باید مستقل باشد و

instansitation هم نباید بکنیم. یا سیستم ما خانواده های مختلفی از محصولات دارد و در هر لحظه باید به یکی از اینها پیکر بندی شود یعنی instance های قبلی را دور بریزیم و خانواده محصولات جدید را میگذاریم. حالت سوم این هست که یک مجموعه خانواده داریم که با هم دارند کار میکنند و میخواهیم مطمئن شویم با هم کار میکنند. حالت چهارم یک سری محصولات داریم که فقط میخواهیم با واسط آنها کار کنیم و داخل آنها را کاری نداشته باشیم و فقط به کلاینت واسط بدهیم. یعنی یک سری زیر سیستم و کلی محصول داریم ولی فقط واسط داریم و بقیه دیده نمیشود. دقت کن کلاینت میداند با ابجکتی که میگیرد چطور کار میکند. دقت کن کلاس های concreate کاملا پوشیده شدند و کلاینت آنها را نمیبیند و فقط واسط میشناسد. جابجایی خانواده محصولات آسان هست.

Builder: میخواهیم ابجکت پیچیده بسازیم ولی فرآیند و الگوریتم ساخت مشترک باشد ولی چیز ساخته شده جنس ها و نوع های مختلفی داشته باشد. مثل تولید ماشین های مختلف. این واسه وقتی هست که الگوریتم ایجاد را میخواهی مستقل کنی از اینکه چه قطعاتی استفاده بشوند برای محصول نهایی و چگونه با هم پیکر بندی شوند. الگوریتم از فرآیند دقیق ساخت جدا میشود. یعنی جزئیات فرآیند ساخت را اجرا میکند این builder یعنی اون هست که میداند که از چه قطعاتی باید ابجکت بگیرد و بسازد طبق دستور director کار میکند ولی خودش مستقل این تصمیمات را میگیرد. این فرآیند ساخت را باید اجازه بدهد که ابجکت ها باید با ساختار های داخلی متنوع ساخته شود چه ساده باشد چه پیچیده عین انواع تولید خودرو یعنی الگوریتم یکسان ولی قطعات و تولیدات مختلفی هستند. هر builder یک director دارد انگار داخل شکم آن هست حالت composition دارد البته ممکن هست stateless و shared شده باشد. دقت کن اینکه builder داخل شکم director باشد لزوما درست نیست اجباری نیست. رابطه بین این ۲ یک association هست دید هم یک سویه هست و پیاده سازی آن آسان تر هست همیشه از director به builder هست چون بیلدر اصلا کاری با اون ندارد سرویسی نمیگیرد. Director قرار نیست محصول را از بیلدر بگیرد چون وابستگی زیادی هست اصلا دیدی نسبت به محصول نهایی ندارد تا وابستگی زیادی نباشد. Director فقط واسط بیلدر را میبیند و زیر کلاس های آن را نمیبیند و OCP برقرار هست. کلاینت ها ولی نیاز هست concreate ها را بشناسند و اونجا وابستگی بالا هست البته میتوان کمتر کرد ولی مرسوم به همین شکل هست. دقت کن بخاطر همین کاهش وابستگی هست که get result داخل کلاس پایینی هست نه خود بیلدر. بیلدر ایجاد ساخت محصولات با جزئیات مختلف میدهد. کد ایجاد و کد ساختار داخلی یا ساخت نهایی از همدیگر مستقل هستند. کنترل ریز دانه به فرآیند ساخت میدهد چون بیلدر گام به گام میسازد و عملیات ها ریز دانه هست و تحت نظر director کنترل میکنیم این کار را.

جلسه هفتم:

Prototype: برای حل مشکل انعطاف پذیری به کار میرود و مشکل factory method را هم حل میکند چون داخل آن DIP حل میکند چون داخل ساختار اون نقض میشد زیرا زیر کلاس های concreate مستقیم با هم در ارتباط هستند و این را بر اساس instance سازی یا cloning انجام میدهد یعنی به جای گرفتن یک ابجکت از یک کلاس یک ابجکت در اختیار اون creator قرار میدهیم در زمان اجرا و هر وقت خواست ابجکت خاصی را بسازد از اون ابجکتی در اختیارش هست یک نمونه کپی میکند یا prototype میزند یعنی به جای گرفتن ابجکت از کلاس از ابجکت فعلی یک کپی میگیرد. برای حل مشکل ساختار توارث موازی و برقراری DIP هست. نمونه اولیه اگر عوض شود رفتار اون هم عوض میشود و نیاز به زیر کلاس ندارد چون هر کاری لازم باشد بخواهد بکند در سمت محصول پیاده سازی شده و وابستگی ندارد و تخصص کار لازم نیست و کلا یک دونه کلاس داریم هر چی بخواهیم سمت خود محصول هست و اون خودش ابجکت میگیرد و کپی میکند. دقت کن خود اون طرف از زیر کلاس های محصول اطلاعی ندارد و وابستگی کم هست. و دقت کن به اون محصول هیچ وابستگی وجود ندارد چون کاربر اصلا محصولات جدید را نمیبیند ارتباط مستقیم ندارد و یک واسط دارد باهاش که فقط از آن ابجکت میگیرد و دچار تغییر رفتار میشود کلاس هایی که ابجکت میگیرند در زمان اجرا مشخص میشوند و از قبل مشخص نیست. یک کاربر این هست که میخواهیم ساختار توارثی از محصولات داشته باشیم ولی به صورت توارثی از سازنده ها نمیخواهیم ایجاد کنیم تا دوباره دچار factory method نشویم و وابستگی concreate به concreate رخ نمیدهد. یک کاربرد دیگر این هست که تنوع حالات برای یک کلاس زیاد نیست و محدود هست و مقادیر عملیات و مقادیر داده آن ها یکسان هست و تنوع ویژگی ها کم هست شاید کپی بگیریم برای ابجکت ها بهتر باشد یعنی برای یک کلاس در ابتدای اجرا سیستم به ازای هر حالت ممکن یک ابجکت میگیرند و ویژگی ها را مقدار دهی میکنند و در زمان اجرا اگر خواستیم کپی بگیرند از یکی از اینها استفاده میکنند و کپی میگیرند. دقت کن در سمت کلاینت به ازای هر ابجکت یک عملیات دارند که اون عملیات کپی میکند و انجام میدهد. این الگو یک خوبی دارد برای کلاس های concreate که اینها از کلاینت ها مخفی هستند یا همون سازنده ها و اصلا به آن پیام نمیدهند و دیدی به آن ندارد و DIP برقرار هست بنابراین OCP برقرار هست. دومین نتیجه این هست که کلاینت ها میتوانند با اپلیکیشن هایی خاص کار بکنند یعنی وابستگی به نوع سیستم خاصی ندارد و هر کلاسی را میتواند باهاش کار کند چون اصلا نیازی به اطلاعات از کلاس های concreate و وابستگی ندارد کلا یک واسط میشناسد و اصلا با نتیجه کار آنها و نحوه کار آنها کاری ندارد و در زمان اجرا این کلاس ها میتوانند کم یا زیاد شوند و تاثیری روی کلاینت ندارد و دچار تغییر نمیشود و فقط یک واسط هست. ما میتوانیم ابجکت و کلاس هایی جدید طراحی کنیم با عوض کردن تغییر مقادیر عین همون تغییر حالات و ترکیب نامتناهی از عناصر داشته باشیم و روی همه اونها کپی بگیریم و هر عنصر مرکب یک کلاس هست و ازش ابجکت میگیریم و این به صورت پویا هست که کلاس میسازیم و سطح بالای انعطاف پذیری هست.

Singleton: این واسه وقتی هست که یک کلاس داریم فقط یک ابجکت داریم و میخواهیم از همه جا قابل دستیابی باشد و روی دستیابی کنترل داشته باشیم ولی نمیخواهیم از یک ابجکت بشود دو تا میخواهیم این را کنترل کنیم. دقت کن نمیتوانیم متغیر سراسری بگذاریم چون به شدت خطرناک هست و وابستگی را بالا میبرد. و این ابجکت داخل کلاس ها و دستیابی فقط داخل کلاس هست یعنی اسکوپ آن و کلاس تضمین میکند تک باشد و داخل کلاس static هست. و این شکلی هم روی دستیابی هم روی تعداد کنترل خوبی دارد. دقت کن به این ابجکت یک گلوبال پوینتر هم داریم. دقت کن اگر قرار باشد که بیای به جای ابجکت عملیات ها را ایستا تعریف کنی کار اشتباهی هست چون قرار نیست کلاس برای ما کار انجام دهد قرار هست ابجکت برای ما کار انجام دهد تو مسئولیت کار دادی به کلاس و ابجکت ها هستند که در زمان اجرا کار میکنند. از این الگو میخواهیم فقط از یک محل دستیابی خاصی دسترسی داشته باشند و فقط یک ابجکت داشته باشد و میتوان این ابجکت را گسترش داد و با استفاده از زیر کلاس گرفتن این ابجکت را گسترش داد و کلاینت ها میتوانند با ابجکت های زیر کلاس ها کار بکنند چون به آنها وابسته نیستند بلکه فقط به کلاس بالایی وابسته هستند. مزایا: دسترسی کنترل شده به تک ابجکت میدهد و از طریق کلاس کنترل میشود و محدودیت دستیابی اعمال میکند. Name space و وابستگی متغیر های سراسری را نخواهیم داشت. امکان میدهد از طریق زیر کلاس گیری ابجکت را گسترش بدهیم و عملیات های جدید بتوانند اضافه کنیم. میتوانیم تعداد ابجکت ها را بیشتر کنیم. از کلاس های ایستا انعطاف پذیر تر هستند و اگر ایستا استفاده کنی زیر کلاس گیری دچار مشکل میشود. ابجکت های facade در سیستم ها یکه هستند البته میتوانند بیشتر باشند. بسیاری از wrapper ها یکه هستند به جز proxy, detector.

الگو های ساختاری:

الگو هایی هستند که بیشتر به ساختار دهی کلاس ها در سیستم ها توجه دارند با هدف کم کردن وابستگی و به این شکل که abstraction از پیاده سازی جدا میکنیم یعنی انتزاع را از پیاده سازی جدا میکنیم و امکان رشد مستقل این ۲ را میدهیم. Wrapper ها همه از این دسته هستند.

Adaptor:

این یکی از wrapper ها هست و دو نوع پیاده سازی دارد یکی از طرف کلاس که با gen/spec دارد یا توارث که در کامپایل تایم محقق میشود و یک شکل دیگر مزایا که بیشتری دارد شکل ابجکت اسکوپ هست که با استفاده از delegation پیاده سازی میشود هدف آن کلا الگو های wrapper به این شکل هست که یک ابجکتی روی ابجکت دیگر میشیند و آن را بسته بندی میکند و نقش او را ایفا میکند برای اجرای عملیات و انگار یک واسط هست روی ابجکت اصلی و کلاینت بیرونی فکر میکند دارند با ابجکت اصلی کار میکنند چون اون ابجکت واسط دارد ادای اون در میاورد برای این الگو واسه وقتی هست که میخواهیم تبدیل واسط ها را به همدیگر داشته باشیم یعنی که ما یک کلاینتی داریم که این کلاینت نمیتواند با کلاس مورد نظر کار بکند و کلاینت یک واسط دیگر را بلد هست در کد خودش و واسط

اون کلاس هم متفاوت هست راه حل این هست به جای دستکاری آن کلاس که پر هزینه هست بیاد حالت مترجم باشد و اون وسط باشد و واسط ها را به همدیگر تبدیل میکند تا مورد انتظار کلاینت باشد. Wrapper adapter فقط ترجمه نمیکند بلکه رفتار هم اضافه میکند واسه همین بهش وفق دهنده یا مبدل میگویند. موارد استفاده یک که گفتیم دو میخواهیم از یک کلاس استفاده مجدد استفاده کنیم که با کلاس هایی که باهاش مرتبط نیستند یا قابل پیش بینی نیستند بتوانند کار کنند ولی واسط آن ها را نمیتواند وفق دهد اگر این نبود باید پیاده سازی کنیم به چندین اسم مختلف که این خوب اشتباه هست با هزاران اسم و هزاران متغیر تکراری. استفاده سوم این هست که از یک زیر کلاس های متعدد استفاده کنیم ولی نمیخواهیم تک به تک واسط آنها را تبدیل کنیم کلا میخواهیم واسط پدر را تغییر بدهیم و فرزندان آن هم اتوماتیک تغییر کند واسط آنها. هر ابجکت از زیر کلاس ها هم بیان اونها هم وفق داده شدن واسط آنها با پیاده سازی های مختلف چون فوق کلاس را adapt کردیم واسط را این برای ابجکت اسکوپ هست برای کلاس اسکوپ این جواب نمیدهد. برای توارث خصوصی با هدف استفاده مجدد رابطه is a برقرار نیست. دقت کن در زمان اجرا فقط adapter هست و پیاده سازی رفتار های adaptee داخلش هست و دقت کن کلاینت با واسط تارگت کار میکند و درخواست میدهد و adapter اون درخواست را پاسخ میدهد این کلاس اسکوپ هست. در ابجکت اسکوپ باید adapter حتما یک ابجکت از adaptee را بشناسد تا جواب کلاینت را بدهد.

جلسه هشتم:

در کلاس اسکوپ واسط adapter منطبق هست با تارگت که کلاینت با آن کار میکند و adaptee متد هاش به adapter به ارث رسیده است. adapter میتواند رفتار اضافه کند چه تو راه حل ابجکت چه تو راه حل کلاسی یعنی رفتاری که داخل adaptee نیست ولی داخل adapter هست رفتار های آن را هم تازه میتواند override کند. دقت کن در واقع adapter واسط adaptee را به واسط تارگت تبدیل کند ولی دقت کن اگر adaptee زیر کلاس های دیگر داشته باشد روی آنها تاثیری ندارد برای آنها باید adapter جدید اضافه بکنیم این در کلاس هست. سهولت overriding هست. در ابجکت اون زیر کلاس های سایر را هم با یک adapter این کار را انجام میدهد. در زمان اجرا کلا از adaptee ابجکت نیاز نداریم در کلاس اسکوپ. تو ابجکت اسکوپ ۳ تا ابجکت در زمان اجرا داریم و کار داده میشود به adaptee دقت کن مثل قبلی override کردن آسان نیست. تو ابجکت اسکوپ تمام ساب کلاس های adaptee در اختیار adaptor قرار میگیرد. adaptor حتما باید با adaptee ابجکتش ارتباط برقرار کند تا الگو برقرار شود. فقط یک ابجکت در اختیار کلاینت قرار میگیرد و اون adapter هست. در ابجکت اسکوپ یک adapter با تعداد زیادی adaptee میتواند کار کند ولی override کردن به این راحتی نیست.

Bridge: این یکی دیگه wrapper نیست و بهش میگیم پل و انتزاع را از پیاده سازی جدا میکنیم یعنی تعریف یک کلاس را از پیاده سازی جدا میکنیم و به صورت مستقل رشد میکنند و پیاده سازی یک کلاس در زمان اجرا میتواند عوض شود. متد ها هر چه قدر ریز دانه باشند تنوع کار ها بیشتر میشود و عملیات های مختلف را میتوان پیاده سازی کرد. الگو ها معمولا performance را پایین میاورند. و سمت سرویس دهنده را میتوانیم سرویس های متنوع بگیریم بخاطر همین جدا سازی. و بهتر هست متد ها ریز دانه تر باشند. شکل پیاده سازی از ذات انتزاع جدا میشود و ارتباط بین فوق کلاس ها هست و پل اونجا قرار دارد و یک ساختار توارثی کلاس ها داریم و یک ساختار توارثی پیاده سازی ها را داریم. یعنی وجه تعریف با وجه پیاده سازی جدا هست و binding به هم ندارند و پیاده سازی میتواند در زمان اجرا تغییر کند مثلا UI را میتوانی تغییر بدهی. Binding در زمان اجرا مشخص میشود نه در زمان کامپایل. ساختار توارثی های موازی را یا با هم ترکیب شده اند را جدا کنید پل بگذار تو فوق کلاس ها. کاربرد: ۱. وقتی میخواهیم یک binding دائمی بین یک انتزاع و پیاده سازی آن وجود نداشته باشد و پرهیز کنیم ذات کلاس جدا پیاده سازی جدا ۲. وقتی میخواهیم انتزاع و پیاده سازی قابل گسترش باشند از طریق ساب کلاس و پیاده سازی های مختلفی داشته باشیم. ۳. میخواهیم پیاده سازی یک انتزاع تاثیری روی کلاینت نداشته باشد یعنی کلاینت فقط پنجره را میبیند و یا واسط را بدون درکی و وابستگی به پیاده سازی و کاملا استقلال وجود دارد. ۴. یک idiom برای سی پلاس پلاس هست. ۵. وقتی میخواهیم پیاده سازی را به اشتراک بگذاریم یعنی ابجکت های مختلف از یک پیاده سازی استفاده کنند و کلاینت متوجه نشود. در زمان اجرا کلاینت فقط یک ابجکت از abstraction میگیرد بعد اون خودش بقیه ابجکت ها را میگیرد. خود abstraction هم یک ابجکت از implementor میگیرد در واقع ۳ تا ابجکت داریم. دقت کن این ابجکت آخری را یک پیکربند باید انجام بدهد که تمام زیر کلاس ها را میسازد و گرنه کلاینت نمیشناسد و نمیتواند. کلاینت وابستگی به ساب کلاس ها ندارد و آنها میتوانند رشد کنند. واسط و پیاده سازی از هم جدا هستند یعنی یک انتزاع را با یک پیاده سازی خاص پیکر بندی کنیم یا حتی عوض کنیم با یکی دیگه در زمان اجرا. گسترش میتوانیم بدهیم و DIP برقرار هست. جزئیات پیاده سازی از کلاینت پنهان هست و کلاینت هیچ دیدی ندارد.

Composite:

در این الگو مواجه هستیم با کلاینتی که میخواهد کار بکند با یک تعدادی عنصر پیچیده و ساده و کلاینت باید با هر ۲ اینها کار کند بدون اینکه تمایزی بین این ۲ قائل شود یعنی از دید اون بتوان یکسان کار کند و فرقی نداشته باشد در واقع transparency باید وجود داشته باشد. کلاینت واسط فوق کلاس را فقط خواهد دید و بین اجزا مرکب یک ارتباط aggregation از نوع composition یا رابطه سخت گیرانه بین آنها هست. اشتباه این شرایط چی هست که این الگو اومد؟ یک رابطه که بین همه مشترک نیست را تو فوق کلاس گذاشتیم ولی این رفتار مال یک عنصر پیچیده هست نه عنصر ساده با این کار ما شفافیت آوردیم ولی safety رو آوردیم پایین چون به همه بچه ها اون رفتاری که

مشترک نیستند را میتوانیم ارسال کنیم. وقتی میخواهیم در قالب کلاس ها ساختار های سلسله مراتبی را باز نمایی کنیم و کلاینت هم شفافیت داشته باشد. این الگو به شدت عمومی هست و عدم ایمنی بخاطر همین ایمنی بودن هست چون اون رفتار جزئی رو اوردی تو بالاترین جا گذاشتی. Composite یعنی اینکه یک رابطه composition بین ساب کلاس و فوق کلاس برقرار شود.

Decorator:

یک wrapper هست و هدفش این هست که بشود به یک ابجکت در زمان اجرا به صورت پویا یک رفتار اضافه یا کم کرد دو راه حل دیگر هم هست به جای این الگو مثلا ۱. یک ابجکت شکل ساده داریم و یک شکل گسترش یافته یعنی یک کلاس با رفتار ساده و یک زیر کلاس با قابلیت اضافی ازش میگیریم و هعی همینطوری ازش ساب کلاس میگیریم رفتار اضافه میکنیم و این اشتباه هست. یا باید یک ابجکت را نابود کنیم ابجکت جدید بسازیم و حالات را بین این ۲ منتقل کنیم که منطقی نیست. این عیب همان مثال دانشجو هست بخاطر توارث. دقت کن وقتی زنجیره شکل گرفت در بازگشت رفتار داریم و کلاینت سر زنجیره هست و فقط سر زنجیره را میبیند. زنجیره در تمامیت در اختیار کلاینت هست نه یک دید محدود. و در زمان اجرا زنجیره را پیکربندی میکنیم و مسئولیت ها در زمان اجرا به صورت پویا به ابجکت ها اضافه میشوند و به صورت شفاف و فقط باید دقت کنید ابجکت ها باید به عنصر ابتدایی اشاره کنند. سیستم منعطف هست. حتی میتوانیم مسئولیت ها را حذف کنیم علاوه بر اضافه کردن برای موقعی که ترکیبات خیلی متعددی از رفتار ها وجود داشته باشند و راه حل بهتری هست نسبت به زیر کلاس گیری. نسبت به توارث ایستا خیلی بهتر هست. تبعات: با on/off کردن ویژگی ها رفتار هاش کم و زیاد میشود ولی این خوب نیست بهش feature made in class گفته میشود و این چیز بد است. و با decorator حل میکنیم تبعات بد: decorator و کامپوننت خودش یکی نیستند یعنی کلاینت برای اینکه ببیند ابجکت در اختیارش عوض شده یا نه و در اینجا ممکن هست ما ابجکت سر زنجیره را میبینم که این هم هعی عوض میشود و فکر میکند ابجکت عوض شده ولی عوض نشده پس نباید از identity استفاده کرد. دومین مسئله این هست که تعداد زیادی ابجکت کوچک داریم یعنی هر ابجکت اصلی یک دنباله ای از ابجکت ریز های بیشتری داریم و سیستم کند میشود کارایی کم شد ولی انعطاف بیشتر شد. Message chain هم یک چیز بد هست.

قسمت نهم

Facade: به معنی نماد هست یا وجه یا دریچه یا منظر در واقع نمای بیرونی یک ساب سیستم هست یک ابجکت هست که روی ساب سیستم نشسته است و نماینده اون ساب سیستم در قالب اون کلاینت ها. وابستگی زیاد هست چون کلاینت ها دید به داخل دارند اگر از این الگو استفاده نکنیم دید داریم و جزئیات کلاس ها رو باید بشناسیم و تغییرات به بیرون منتشر شود بخاطر همین وابستگی زیاد هست بخاطر همین از این الگو استفاده میکنیم و یک واسط تعریف میکنیم تا از این ساب سیستم بتوانیم استفاده کنیم. کلاینت ها با این واسط دیگر دید به داخل ندارند ولی این facade پیچیده

میشود دقت کن صرفا فقط این یک واسط هست و کار را رو اداره میکند نیاز ندارد کاری را پیاده سازی کند نباید مسئولیتی به آن داده شود صرفا فقط باید کار را اداره کند و بگرداند همین. Facade درست هست که واسط هست ولی در واقع یک ابجکت هست. وابستگی یعنی از تغییرات آن دچار تغییر شوند. Generality نداریم چون دیگر کلاینت داخل ساب سیستم دید ندارد صرفا ease of use دارد این الگو و نباید اجازه بدیم به generality تعریف شود و اگر کلاینتی ناقص بود باید بیای و واسط رو بهتر کنی نه اینکه به داخل دید برقرار کنی. ممکن هست از یک کلاس facade چند تا ابجکت داشته باشیم و لزومی ندارد single tone باشد ولی معمولا این شکلی هست. با همین میتوانیم دید کلاس های پیاده سازی را از کلاینت ها مخفی کنیم. برای لایه بندی سیستم هم مورد نیاز هست که هر لایه به لایه دیگر دید ندارد. مزیت این هست که کلاینت به اجزای داخلی وابسته نیستند. دقت کن با فساد دیگر دسترسی مستقیم نداریم.

Flyweight:

در بعضی از سیستم ها اینقدر تعداد ابجکت زیاد شود که دنبال شی گرایی نرویم. یک سری ابجکت ریز دانه داریم و یک حوض از آنها داریم و اشتراک گذاری داریم یعنی به ازای کل a ها داخل متن مثلا فقط یک رفرنس درون این استخر داریم و ابجکت کم شد و مزایا شی گرایی هست منتها یک مشکلی دارد این هست که خود کد کاراکتر a که داخل این حوض هست یک سری اطلاعات دیگر دارد که دیگر نمیشود درون گذاشت مثلا اینکه چه فونتی داشته باشد یا در چه مکانی باشی در چه ستونی در چه ردیفی ولی الان دیگر نمیتوانیم همه این اطلاعات را بگذاریم درون این ابجکت قبلا خودش میدونست کجا با چه فونتی قرار بگیرد و از سایر جا ها دیگه باید تغذیه بشوند و حالت یک ابجکت را به دو قسمت تقسیم میکنیم یک ذاتی که مشترک هست دو که اختصاصی هست و مشترک نیست و باید بقیه ابجکت ها کمک کنند. پس در سیستم هایی استفاده میکنیم که تعداد ابجکت خیلی زیاد هست و هزینه ذخیره سازی بالاست و از اشتراک گذاری کمک میگیریم. در واقع داریم بیشتر حالت ابجکت را به دو قسمت تقسیم میکنیم و اون قسمت مشترک رو نگه میداریم فقط تا کمتر شود. میتوانیم تعداد زیادی از ابجکت ها را با تعداد کمی ابجکت مشترک تعویض کنیم مثلا همین که کلا با تعداد ابجکت به اندازه کاراکتر های انگلیسی جا به جا کردیم البته شرط این هم این است که از identity استفاده نکنیم تا متوجه شویم که این یک ابجکت نیست و ابجکت های متفاوت هستند ولی چون به یک جا رفرنس دارند فکر میکنیم فقط یکی هست که این درست نیست. پایان قسمت ۹.

قسمت دهم:

با کد اسکی در واقع اون حروف را به اشتراک میگذاریم. و برای ابجکت مثلا A نمیتوانیم رنگ و فونت را درون همین بگذاریم در واقع حالات غیر مشترک را باید از بیرون بگیریم. دقت کن واسه فونت اینا ما فقط change ها را نگه میداریم نه کل اطلاعات را. وقتی استفاده میشود که سیستم تعداد ابجکت ها بسیار زیاد هست و برای ذخیره سازی به شدت دچار مشکل میشویم.

از هر ابجکت باید فقط یک نمونه باشد برای همین باید کنترل بکنیمش توسط flywightfactory که همان کنترلر هستش بر روی پول ابجکت ها. در زمان اجرا کلاینت ها سراغ همین کنترلر ها میان و دید پیدا میکنن به پول. صرفه جویی در حافظه دارد ولی بدی آن این هست که محاسبه کردن و بدست آوردن اون ویژگی هایی که مشترک نیستند مثل رنگ و فونت باعث دردسر هست چون همه چیز را نمیتوان به اشتراک گذاشت و هزینه آن بسیار زیاد هست. این الگو در موارد خاص هست و عمومی نیست مثل wrapper ها.

Proxy:

این هم یکی دیگر از wrapper ها هست. مثل decorator adapter facade. این در واقع یک نائب و قائم مقامی برای ابجکت اصلی هست انگار وکیل او هست کلاینت هیچ تفاوتی بین اینها قائل نیست و فکر میکند ابجکت اصلی هست این نائب در واقع یک سری کار ها را برای ابجکت اصلی انجام میدهد یک کار اضافی مثلا برای کنترل دستیابی یا صرفه جویی در استفاده از فضای حافظه و ... ، انواع مختلف دارند و کلاینت تفاوتی بین این ۲ ابجکت قائل نیست. مثلا اگر یه ابجکت سنگین و نیاز به منابع زیاد داشته باشیم بار گیری آنها زودتر از موعد باعث کند شدن یا صدمه سیستم میشود و از پراکسی استفاده میکنیم تا این کار را به تاخیر بیندازیم و تا موقعی که نیاز نبود از پراکسی استفاده میکنیم تا ادای اون ابجکت را در بیاورد مثلا اون رو به عنوان عکس میدیدند ولی وقتی دیگر واقعا نیاز شد میسازد. و وقتی ابجکت نهایی مجبور شد ظاهر شود دیگر فقط کار را رد میکند و دیگر کاری ندارد. این الگو واسه وقتی هست که برای یک ابجکت نیاز به نائب داریم. این چیزی که در جملات قبلی توضیح دادیم virtual proxy بود نوع ها دیگه هم هست، یکی دیگه remote proxy هست، وقتی هست که ابجکت اصلی ریموت هست یعنی ابجکت لوکال نیست مثلا رو یک نود دیگه هست و برای آن در لوکال یک نائب در نظر میگیریم و سایر ابجکت ها فکر میکنند هم این ابجکت اصلی هست هم لوکال هست و هر چی کار داشتند اون نائب میره ارتباط برقرار میکند و سرویس میگیرد میدهد به کلاینت ها و دیگر نیازی به رابطه ریموت ندارند. Protextion proxy: دسترسی را به ابجکت اصلی کنترل میکند به بعضی کلاینت اجازه میدهد به بعضی اجازه نمیدهد این ۳ تا انواع اصلی هستند چهارمی smart refrence هست که پیشرفته حالت پوینتر هست و وقتی کلاینتی با ابجکت اصلی کار نداشته باشد آنرا آزاد میکند به این reference counting گفته میشود دومی lazy loading هست برای اینکه بیاری تو حافظه اصلی به ازای اولین درخواست مثل virtual نیست اون تا موقعی که نیاز نباشد ابجکت را نمیاورد سعی میکند خودش پاسخ دهد ادای آن را در نمیاورد و پاسخگو نیست این lazy loading مورد سوم locking هست برای وقتی که به یک ابجکت همزمان چند تا دیگه نمیتوانند وصل شوند و قفل میکنند. پراکسی ساب کلاس همون واسطی هست که با ابجکت اصلی هم ساب آن هست، قرار دارد و کلاینت واسط را میبیند. پراکسی در بعضی مواقع مثل virtual ابجکت میسازد ولی در سایر موارد پراکسی ابجکت اصلی را نمیسازد.

دقت کن بین پراکسی و ابجکت اصلی باید یک رابطه association باشد تا ارتباط بین آنها حفظ شود رفرنس بین آنها و دید مانا باید برقرار شود. خط چین یعنی دید حفظ نمیشود و نا مانا هست ولی خط دید مانا هست و باید خط بگذاریم چون بعدا باهاش کار داریم.

الگو های رفتاری:

دو دسته الگو داریم اول کلاسی ها در زمان کامپایل و توارث و gen/spec پیاده سازی میشوند که interpreter, template method این الگو از این دسته هستند. دسته دوم ابجکت اسکوپ هستند در زمان اجرا و با استفاده از delegation پیاده سازی میشوند.

Chain of responsibility:

هدف این هست که فرستنده درخواست از گیرنده درخواست را جدا کنیم تا وابستگی کم شود به تعداد زیادی از ابجکت این مجال را میدهیم که این درخواست را دریافت کنند و هر کدام توانستند انجام دهند و پاسخ میدهند گیرنده به صورت پویا انتخاب میشود و ما از قبل نمیدانیم کدام ابجکت این درخواست را جواب میدهد و این شکلی وابستگی تا حدودی کمتر میشود. به نوعی زنجیره بی مسئولیتی هم هست چون همه ابجکت ها کار را به دیگری محول میکنند و میفتد بیرون برای حل این مشکل یک دیفالت help handler (البته این خودش رد کردن درخواست به عنصر بعدی را دارد فقط و ربطی به این بحث ندارد) نام دارد، ته زنجیره هندل میکند تا کار نیفتد بیرون بخاطر این هست که ما کسی را به صورت مستقیم مسئول نکردیم و یک مجموعه را مسئول کردیم.

جلسه یازدهم:

موارد استفاده: وقتی پاسخگویان یک درخواست بیش از یک ابجکت هست و هندلر نهایی در زمان اجرا به صورت پویا مشخص میشود و از قبل تعیین نمیشود. حالت دوم واسه وقتی هست که میخواهی یک درخواست را به یک مجموعه ابجکت بفرستی ولی دریافت کننده را نمیخواهی مشخص کنی و قابل تغییر و پویا میخواهی باشد تا شکل پاسخگویی عوض شود. حالت سوم مجموعه پاسخگو ها به صورت پویا میخواهیم تعیین شود یعنی خود عناصر ممکن هست در زمان اجرا تعیین بشوند. یا مثلا پاسخگو نهایی را به صورت پویا عوض بکنیم. وابستگی کم میشود مسئولیت ها را میتوان در زمان اجرا تغییر داد و پیکربندی را میتوان عوض کرد البته گیرنده گارانتی گرفتن مسئولیت را نمیدهد و درخواست پاسخ داده نشود و هیچ گیرنده ای نپذیرد.

Command:

هدف این الگو هست که پیغام یا درخواست یک ابجکت به ابجکت دیگر را خودش به شکل ابجکت در بیاوریم، یعنی به پیام ها و درخواست ها هویت میدهیم چون انتقال پیغام یک کار آنی هست و هویت خاصی ندارد با ابجکت این هویت پیدا

میکند و ماندگار خواهد شد و با هر ابجکت میتوان باهاش کار کرد و میشود در صف گذاشت و به ترتیب خاصی ارسال کرد یا لاگ کنیم و اگر سیستم کرش کرد به ترتیب لاگ ارسال کنیم و ... ، که بخاطر ماندگار کردن پیغام ها اتفاق میفتد انگار به پیام ها هویت میدهیم و امکان ذخیره کردن و لود کردن و ارسال به ترتیب خاص دارند میتوان undo را با همین پیاده سازی کرد و به خود ابجکت میگیم برگردان به حالت قبل چون حالت قبلی را نگه میدارند و وقتی undo اومد برمیگردیم عقب. یعنی اون پیام به کامند متناظر میگوید execute و کامند بین این ۲ ابجکت اون هست که کار را انجام میدهد و ابجکت کامند به اون ابجکت الف پیکر بندی میشود و به آن داده میشود حتی این کامند میتواند عوض شود. macro command داریم یعنی یک کامند که خودش از چند تا کامند دیگر تشکیل شده است و اون سر کامند کار را به کامند های زیرین خودش تحویل میدهد. پس میتوانیم پارامتری کنیم ابجکت الف را تا با کامند های مختلف کار کند و در زمان اجرا پیکر بندی میشود و نحوه اجرای عمل مشخص میشود. در واقع به جای ارسال پیغام یک درخواست اجرا میخواهیم که این کاربرد اول هست و ابجکت با عملیات پیکربندی میشود در زمان اجرا. مزیت دوم استفاده از صف این هست که درخواست ها در زمان اجرا به صف میکنیم و متناسب با زمان میفرستیم یا ترتیب خاص و پیغام اجرا میفرستیم اجرا شوند مزیت سوم همان مزیت undo هست. مزیت چهارم این هست که میتوانیم تغییرات را لاگ کنیم، مزیت پنجم همان macro command هست یعنی عملیات های سیستم را از یک سری عملیات ریز دانه که یک عملیات درشت دانه را شکل میدهند استفاده کنیم و جای ارسال پیغام کامند بفرستیم و این قدرت زیادی دارد و اجرای کامند ها عملیات ها محقق میشود نه ارسال پیام. کامند حتی کاهش وابستگی میدهد چون دیگر بین الف و ب ابجکت ارتباط مستقیم نیست و فرستنده نمیداند گیرنده کیست و فقط میداند باید بگوید اجرا و دستور اجرا میدهد و خود اون ابجکت کامند کار را انجام میدهد. invoker اون دستور اجرا را میزند. کلاینت receiver را میشناسد و با کامند آن را پیکر بندی میکند و invoker هم کلاینت کامند را بهش میدهد تا مشخص شود invoker به کدوم کامند پیغام اجرا بزند و بعد کامند اکشن را ارسال میکند به receiver. نه کلاینت نه invoker نمیدانند receiver کی هستند. وابستگی غیر مستقیم میشود و کامند فقط مقصد را میداند و فرستنده نمیداند در نتیجه وابستگی کم هست، مزیت بعدی این هست که کامند ها ابجکت هستند و با هویت در یک دوره ای وجود دارند، پیکربند ها به ازای تغییرات جدید کامند فقط تغییر میکنند ولی بقیه جا ها عوض نمیشوند هیچ invoker یا ... تغییر نمیکنند.

Iterator:

برا این هست که یک عنصر مرکب یا ابجکت مرکب را عناصر آن را پیمایش کنیم بدون مشخص شدن جزئیات ساختار داخلی آنها مثلا یک لیست داریم و توش پیمایش کنیم، و پیمایش رو به عهده خود ابجکت اصلی نمیگذاریم تا سنگین نشود واسه همین به خود ابجکت مرکب نمیتوانیم بدهیم چون سنگین میشود به کلاینت هم نمیتوانیم بدهیم چون encapsulation نقض میشود و باید دید داشته باشد واسه همین list iterator داریم که متخصص پیمایش

هست این ابجکت و توسط ابجکت مرکب تولید میشود و به ابجکت کلاینت داده میشود اون دستور میدهد چطوری حرکت کند مثلا برو جلو برو خونه اول و ... ، و encapsulation حفظ میشود به عنصر مرکب هم سپرده نشده و به این ابجکت متخصص پیمایش کننده سپرده میشود و جلو میرود و دقت کن این مرکب هست که ابجکت متخصص پیمایش میسازد و به کلاینت میدهد و بعد به طور همزمان میتوانیم چند ابجکت متخصص پیمایش گر داشته باشیم و چند پیمایش صورت بگیرد و وضعیت پیمایش داخل همین متخصص هست. البته وابستگی بین متخصص و ابجکت مرکب زیاد میشود ولی چاره ای نیست تا اون ابجکت ساده بماند. پس شد ۱. کپسول سازی واسه کلاینت نقض نشود و ساختار درونی نمایش داده نشود ۲. چندین نوع پیمایش مختلف داشته باشیم ۳. میخواهیم ساختار های پیچیده مختلف را با واسط واحد پیمایش دهیم.

پایان قسمت ۱۱.

جلسه دوازدهم:

Mediator:

این الگو پر کاربرد و خطرناک هست در این الگو در وضعیت نامطلوب یک تعداد ابجکت داریم که به شکل در هم تنیده با هم کار میکنند و وابستگی بالاست چون به شدت به هم دید دارند راه حل این هست که به عنوان یک واسطه یک ابجکت قرار میدهیم و ارتباط بین ابجکت ها را قطع میکنند و ابجکت ها دیگر همدیگر را نمیشناسند و از هم سرویس نمیگیرند و دید ندارند و هر کاری داشتن فقط به واسطه میدهند و به شدت مطلوب هست و وابستگی میاد پایین. و این واسط دستور میدهد هر کسی چه کاری انجام بدهد و سایرین فقط درخواست میفرستند به نوعی نماینده کل این مجموعه هست. به نوعی همون facade هست چون برای کلاینت بیرونی هم یک واسط هست. فساد بحث کلاینت با یک سیستم پیچیده بود ولی اینجا وابستگی صرفا بین ابجکت ها بالاست. ارتباطات را غیر مستقیم میکند از بیرون هم غیر مستقیم هست و وابستگی کم میشود. هر widget یک director دارد و هر کدام باید متناظر خود را ببینند. دقت کن اونایی که فقط سرویس دهنده هستند و اونها واسطه را نمیبینند یا میانجی رو فقط میانجی اونها را میبینند و از آنها سرویس میگیرند جز همکاران نیستند و خطری ندارند و جز درهمتنیدگی نیستند. میانجی کلا جایی هست که کار کردن این ابجکت ها به صورت پیچیده هست و انتشار تغییرات به صورت گسترده هست و وابستگی به شدت بالا هست اینجا از میانجی استفاده میکنیم، یک کاربرد دیگر این هست که ما یک رفتار توزیع شده و جمعی را گسترش بدهیم یک رفتار تعاملی جمع که به مجموعه ابجکت ها هست وقتی بخواهیم گسترش بدهیم باید همه ابجکت ها را گسترش بدهیم ولی با این الگو دیگر نیازی به این کار نیست. حسن ها: ساب کلاس ها را محدود میکند و رفتار های توزیع شده را مرکز میدهد و تغییر و گسترش رفتار بسیار راحت میشود. همکار ها ارتباط غیر مستقیم دارند و وابستگی کم میشود. پروتکول ها به شدت آسان میشود چون ابجکت ها با میانجی فقط کار میکنند و دیگر پیاده سازی سرویس گرفتن از دیگر ابجکت ها را ندارند. تعامل توزیع شده فقط داخل یک ابجکت میگیرد و ابجکت فقط کار تخصصی صورت میگیرد. و ابجکت ها cohesion آنها بالاتر میرود.

میانجی متخصص تعامل بین همکاران هست و همکاران فقط تخصص کار خودشان را دارند. و اگر میخواستی تعامل و رفتار را عوض کنی میدانی فقط باید سراغ میانجی بروی. بدی هست که این مرکز گرایی میتواند پیچیده شود.

Chapter 5:

Memento:

که میشود یادگاری یعنی هر چیزی که میتواند کمک به یاد آوردن چیزی، وقتی استفاده میشود که میخواهیم حالت داخلی یک ابجکت را ذخیره کنیم برای اینکه بعدا برش گردانیم به همین حالت برای undo کردن استفاده میشود اگر این را بگذاریم به عهده خود ابجکت بیخودی سنگین میشود چون حالات خودش را داخل خودش ذخیره میکند و نحوه ذخیره سازی هم داخل خودش هست و از نظر رفتاری هم سنگین میشود در صورتی که این یک کاربرد بیرونی هست کلاینت نیاز دارد بداند با کدام حالت کار کند نه اینکه خود ابجکت بداند واسه همین از یک ابجکت به نام memento استفاده میکند که ابجکت اصلی میسازد هنگامی که نیاز داشت و کلاینت در خواست داد و بعد از ساخت میدهد به خود کلاینت میگوید تو کنترل کن این ابجکت را و میگوید حالات خودت را بر اساس memento ذخیره کن تا بعدا به اون حالت بتوانیم برگردیم و همچنین نباید محتویات داخلی در معرض دید کلاینت ها باشد بخاطر همین یک ابجکت میگیریم که حالت را دارند ولی داخل اون جزئیات پیاده سازی آن را کلاینت ها نمیبینند و encapsulate میشود حالت هم مجموعه مقادیر ویژگی ها به همراه سایر ابجکت هایی هست که میشناسد. بعد از درخواست set memento اون ابجکت اصلی این را میگیرد و حالت خودش را بر اساس این یادآور تنظیم میکند. یک واسط narrow به کلاینت و یک wide واسط به ابجکت اصلی تا دید داخلی نسبت به جزئیات حالات اون ابجکت کلاینت نداشته باشد. ابجکت اصلی هست میاد حالت رو تغییر میدهد اون memento فقط ذخیره میکند. تبعات: این الگو کلاینت ها را از دید به داخل و جزئیات ابجکت اصلی محفوظ نگه میدارد و encapsule میشود. وابستگی بین ابجکت اصلی و memento زیاد میشود چون ابجکت اصلی باید دید به داخل داشته باشد ولی مهم نیست مهم از نظر کلاینت هست. در نتیجه این الگو ابجکت اصلی سبک میشود و مسئولیت نگه داری حالات به عهده کلاینت ها هست. تبعات بد: استفاده از این الگو در عمل پر هزینه هست چون مجموعه بزرگی از حالات را باید ذخیره کنیم و این ها را باید به عهده کلاینت بگذاریم و این بار سنگین اضافی هست، چون جای معکوس کردن عملیات حالات را معکوس میکنند. رفتار ایجاد کردن این ابجکت یاد آور هم وقت گیر و زمان بر هست. ممکن هست کلاینت ها زود تر دستور ساخت memento بفرستند و زمان بر هست پس کلا استفاده از این الگو سنگین هست مورد بعدی این هست که دو نوع واسط باید تعریف بکنیم و در بعضی زبان ها دچار مشکل میشود. مورد بعدی این هست که هزینه مخفی هم داریم بخاطر مسئله garbage collection.

Observer:

یه چی داشتیم به نام broadcast که انتشار تغییرات بین ساب سیستم ها بود که میخواستیم با حداقل وابستگی ایجاد شود ولی این یه چیز خاص هست، اینجا یک ابجکتی داریم که نسبت به تغییرات حساسیت داریم ولی میخواهیم انتشار تغییرات آن با کمترین وابستگی باشد ما یک subject داریم و یک سری observer که حساس هستند به تغییر سابجکت، ممکن هست یک observer بتواند سابجکت را عوض کند اصلا ولی یک سری فقط ناظر باشند بدون تغییر. سابجکت فقط باید به ناظر ها اپدیت بفرستد فقط میداند باید پیام اپدیت بفرستد بیشتر از آن چیزی نمیدانند حتی نمیدانند به کدام تغییر حساس هستند یا خودشان را اپدیت میکنند یا نه فقط پیام اپدیت میفرستند. سابجکت فقط فرض میکند همه به تغییر حساس هستند و پیام میفرستد همین حتی ماهیت تغییر و پارامتر اپدیت فرستاده نمیشود فقط میگیم اپدیت. بخاطر همین وابستگی به شدت کم میشود. موارد استفاده: یک کلاس دو وجه در تعامل با هم دارد و اگر به ۲ کلاس شکسته شود وابستگی به شدت زیاد میشود و پایین آمدن استفاده مجدد هست ولی اگر ۲ وجه مختلف و تغییراتی که با هم دارند را با استفاده از این الگو پیاده سازی بکنیم این مشکلات را ندارند. با این الگو تغییرات اضافی را ما منتشر نمیکنیم اصلا. سابجکت فقط با اپدیت کار دارد همین. با attach ناظر ها ثبت نام میکنند و میروند در لیستی که اپدیت باید بگیرند notify به تمام لیست پیام اپدیت میدهد هر ناظری نخواست میاد خودش را detach میکند. دقت کن تمام ناظر ها نیاز ندارند سابجکت را بشناسند و دید داشته باشند صرفا همینکه بدانند تغییر هست کافی هست و نیاز به get state , set state ندارند. Set state فقط واسه بعضی ناظر ها هست ممکن هست بعضی ناظر ها نه set, get state بکنند ممکن هست بعضی ناظر ها فقط get state بکنند.

قسمت سیزدهم:

ادامه الگو قبلی: ارتباط از نوع broadcast به خوبی برقرار میشود. عواقب بد: ممکن است اپدیت های غیر منتظره در سیستم رخ بدهد و کارایی سیستم بیاد پایین چون ناظر ها از همدیگر خبری ندارند و بیان set state کنند و نتایج منفی به بار میاورد چون سابجکت هر بار تغییر میکند کلی پیام اپدیت میفرستد با هر تغییر کوچک و کلی پیغام اپدیت برای تعداد زیادی ناظر ارسال میکنیم و کارایی میاد پایین اتفاق دیگر بعضی از این ناظر ها خودشان سابجکت هستند برای ناظر های دیگه یعنی وقتی خودشون تغییر میکنند باید پیغام اپدیت بفرستند و استفاده بی مورد و set state الکی کارایی را به شدت پایین میاورد و باید set state حداقلی باشند.

State:

این پرکار برد هست عین همون ۷ تا الگو کد هست به نام roles played و اینجا میخواهیم یک ابجکت رفتار وابسته به حالت داشته باشیم و هر موقع حالت عوض شد رفتار هم عوض شد انگار که کلاس اون ابجکت عوض شده است ولی

در واقع اینطوری نیست در واقع اون حالت داخلی عوض شده است و ابجکت اصلی باید ابجکت حالت را درون خود حفظ بکند و هر موقع حالت عوض شد ابجکت داخلی عوض بشود و رفتار هم تغییر بکند مثل همان وضعیت دانشجو. برای حالات متغیر باید استفاده بکنی به صورت اتوماتیک هم باید رفتار عوض بشود. و چون ابجکت اصلی این وظیفه را delegate کرده مطمئنیم با تغییر حالت رفتار هم عوض میشود. اون بخش کار وابسته به حالت داده میشود به ابجکت داخلی مسئول نگه داشتن حالت. موارد استفاده: یک ابجکتی که رفتار وابسته به حالت هست و در زمان اجرا با عوض شدن حالت باید عوض شود. مورد بعدی operation ها سوییچ سنگین وابسته به حالت ابجکت دارند و بر اساس حالت انتخاب میشود دقت کن هر تغییر مقدار باعث تغییر حالت نمیشود مثل موجودی بانک. مکرر باید چک کنیم آیا با تغییر یک ویژگی باعث تغییر وضعیت و تغییر حالت میشود یا نه و پیچیده هست. دقت کن خود ابجکت حالت باید بداند به حالت بعدی باید برود یا نه و یک نمونه از حالت بعدی میگیرد و context رو پیکر بندی میکند با آن برعکس نباید باشد چون وابستگی زیاد میشود دقت کن خود context باید واسط داشته باشد تا ابجکت حالت بتواند پیکر بندی بکند آن را. نتایج: رفتار وابسته به حالت را جمع میکنیم به ابجکت حالت مربوطه و هر رفتار به ابجکت حالت آن مربوط میشود و ابجکت پیچیده نمیشود. جابجایی حالت به صورت صریح رخ میدهد. ابجکت های حالات میتوانند به اشتراک گذاشته شوند چون در زمان اجرا ابجکت حالات مختلفی داریم و تعداد زیاد هست ولی اگر اشتراک گذاری کنیم تا سرویس بدهند به context های مختلف این مشکل حل میشود و دقت کن تغییر حالت به عهده context نیست به عهده خود ابجکت حالت هست.

Strategy:

این الگو شبیه قبلی هست و ما یک خانواده از الگوریتم داریم و بر اساس شرایط میخواهیم یکی انجام شود مثل از نظر فضا و زمان بهینه هستند بر اساس شرایط یا بر اساس نیاز کلاینت و ...، میخواهیم الگوریتم را در زمان اجرا جایگزین کنیم بدون وابستگی اضافی و سیستم بتوان پیکربندی کرد و خود الگوریتم را بتوان بسط داد بدون تغییر در جای دیگه. دقت کن composition یک دید یک طرفه به compositor دارد که اون الگوریتم ها را دارد و زیر کلاس اون هستند و به الگوریتم ها composition دیدی ندارند. بدی این هست که پیکربند یا کلاینت نیاز دارد تمام الگوریتم های زیر کلاس را بشناسند و در اختیار composition قرار بدهند و این وابستگی اضافی هست ولی چاره ای نیست. موارد استفاده: تعدادی زیادی کلاس داریم که فقط در رفتار و پیاده سازی رفتار با هم متفاوت هستند یعنی حتی واسط هم یکسان هست و این امکان را میدهد که بشود یک کلاس داشت و رفتار را پیکربندی میکنیم به اون کلاس و واسط واحد و به ازای هر ابجکت با یک رفتار پیکربندی میشود. کاربرد بعدی که مهم هست همان نیاز به الگوریتم های مختلف هست که در زمان اجرا عوض بکنیم. مورد بعدی این هست که الگوریتم از داده های استفاده میکند که کلاینت نباید چیزی از آن بداند اگر بزاریم در اصلی سنگین میشود ولی اگر در الگوریتم استراتژی بگذاریم دید از بیرون پنهان میشود و سبک میشود. چون الگوریتم را جدا کردیم و داخل استراتژی گذاشتیم سبک میشود.

در مورد بعدی این هست که یک کلاس تعداد زیادی رفتار از خودش بروز میدهد که اینها خودشان را به صورت سوییچ در عملیات ها خودشان را نشان میدهند این رو تبدیل کنیم به استراتژی همین رفتار بخش بندی میشود در زیر کلاس های آن و دیگر پیچیده نیست دیگر و سوییچ از بین میرود دقت کن استراتژی در سطح متد میتواند اصلا باشد ولی رفتار میتواند بیش از یک عملیات را در بر بگیرد. بر خلاف state که بر اساس شرایط و حالات و برای خود context هستند اینجوری اینجا نیست که بر اساس شرایط و حالات باشد و میتواند اصلا بیرون ابجکت یا ترجیح کاربر یا شرایط سیستمی تعیین کند استراتژی را ولی شکل ساختاری آنها به شدت یک شکل هست. یک مشکل هست که واسط کلاس استراتژی پیچیده ترین حالات و داده را در نظر میگیرد ولو فقط یکی از اون الگوریتم ها پیچیده باشد و بقیه ساده باشند و این بد هست چون پارامتر برای همه اینها رد میشود در صورتی که نیاز نیست ولی چاره ای نیست چون واسط باید کلی باشد راجب تمامی ساب کلاس ها. یک راه دیگر به جای رد کردن پارامتر این هست مثل الگو قبلی context خودش را معرفی کند تا ابجکت و مقادیر آن استفاده کند. عواقب: اجازه میدهد خانواده ای از الگوریتم ها مختلف بسازیم. یک جایگزینی از زیر کلاس گیری خواهد بود چون دیگر لازم نیست برای هر عملیات یک زیر کلاس از context بگیریم مثل مثال دانشجو و اون ساختار درختی ترکیبی هعی زیاد میشود و رفتار زیر کلاسی مطلوب نیست و استراتژی delegation هست و بهتر هست. برای تغییر استراتژی هم لازم نیست ابجکت را بکشی مثل مثال دانشجو. استراتژی سوییچ را میتواند از بین ببرد و لاجیک کد ساده شود. مورد آخر استفاده از الگوریتم مختلف هست بر اساس شرایط که یک کار و یک نتیجه یکسان میدهند ولی یکی زمان بهینه هست یکی فضا بهینه و بر اساس شرایط سیستم انتخاب میکنیم. بدی: درست هست DIP بین context و استراتژی داریم ولی کلاینت شما کسی که پیکربندی میکند باید زیر کلاس های استراتژی بشناسد و رفتار آن ها را بشناسد و وابستگی بالا میرود و چاره ای هم نیست. بدی این هست که تعامل و ارسال پارامتر زیاد میشود و سنگین میکند سیستم را. مورد بعدی این هست که تعداد ابجکت ها بسیار زیاد هست مانند decorator.

Visitor:

یک عملیاتی داریم که میخواهیم روی عناصر مختلف ساختار ابجکتی اجرا شود اون operation ما. یعنی میخواهیم روی المان های مختلف آن عملیات انجام بدهیم. عملیات ها قبلش این شکلی هست که تو کلاس بالایی میاریم بعد تو پایینی ها هم باید بیاریم خوب این انتشار تغییرات گسترده هست و به ازای هر تغییر هم تو کلاس پدر هم تو زیر کلاس ها تغییر باید اعمال شود به ازای عمل جدید و همه تغییر میکنند. ما visitor داریم که متخصص انجام عملیات هست و به جای اینکه بگیم نود اون عمل جدید را بپذیرد بهش میگیم هد visitor را بپذیر و ما به بازدید کننده عملیات را از آنها میخواهیم انجام بدهد و ابجکت را هم معرفی میکنیم و این روی بازدید کننده ها تعریف شده است. و به اون نود پیام accept میفرستیم و هر نود بر اساس نیاز های خودش روی بازدید کننده اعمال میکند. ولی کپسول سازی نقض میشود چون تمام دید داخل را به بازدید کننده میدهیم و چاره ای هم نیست.

DIP بپرس.

پایان قسمت ۱۳.

قسمت ۱۴:

دقت کن درست هست که تو واسط بالا همه پیاده میشوند ولی هر کدام رفتار متفاوت دارند در زیر کلاس ها و رفتار ها نسبت بی ارتباط به همدیگه هست ولی داخل یک نود هستند و انگار به نوعی از cohesion افتاده هست واسه همین از بازدید کننده استفاده میکنیم. ابجکت command یک نوع ابجکت متخصص هست مثلا برای انجام کار خاص یا استراتژی برای یک کار خاص. انجام توسط یک متخصص خارجی هست و اون هست که دید دارد و باید به اون نود اون بازدید کننده را بهش ارسال کنیم و توسعه عملیات آسان هست چون هیچ دیدی به بازدید کننده نود ها ندارند فقط یک واسط هست و DIP برقرار هست. ولی بر اضافه کردن نود جدید باید داخل نود ویزیتور هم اضافه کنیم یعنی به ازای اضافه کردن نود وزیتور هم تغییر میکند و تغییر گسترش میابد ولی برای بسط دادن عملیات مشکلی نداریم پس بهتر هست عملیات اضافه کنیم نه نود جدید. موارد استفاده: از این الگو وقتی استفاده میشود که ما یک ساختار ابجکتی داریم که تعداد زیادی ابجکت از کلاس های مختلف با واسط مختلف نود هایی تشکیل میدهند و میخواهیم عملیات هایی روی این ساختار ابجکت ها پیمایش کنیم و به کلاس آنها بستگی دارد پس یک واسط کلی نود میخواهیم و هر عملیات را در زیر کلاس ها تعریف کنیم. کلاس ها و ابجکت های آنها تنها اشتراکی دارند این هست که اجزا یک ساختار ابجکتی هستند ولی پیاده سازی های مختلف با واسط های مختلف داریم و باید برای آن ساختار ابجکتی توی همه اینها تعریف کنیم و سنگین میشوند خوب به جای این المان ها را متخصص انجام عملیات پیمایش بکنیم درون بازدید کننده ها تعریف میکنیم. یک مورد دیگر استفاده این هست که تعداد عملیات زیادی که هیچ ربطی بهم ندارند و روی این ساختار ابجکتی میخواهند انجام شوند و نمیخواهیم کلاس های اجزا را به اجرای عملیات وابسته کنیم و المان ها ربطی بهم ندارند و هویت جدا دارند و نباید با عمل به عملیات های آنها را آلوده کنیم. سومین مورد کلاس هایی که یک ساختار ابجکتی را تعریف میکنند خودشان به ندرت عوض میشوند ولی عملیات ها در حال رشد هستند واسه همین به بازدید کننده نیاز داریم. کلاینت ساختار ابجکت را برای پیمایش میبیند و visitor هم میبیند. بازدید کننده زیر کلاس های خود را میشناسد و میداند هر کدام چه عملیاتی را انجام میدهند و در موقع لزوم باید نمونه بگیرد از اینها و بدهد به المان ها. المان ها هم بازدید کننده را قبول میکنند و به المان های زیرین خود و متد مخصوص خود را روی بازدید کننده اعمال میکنند دقت کن ارتباط المان با بازدید کننده از طریق واسط هست و از زیر کلاس های همدیگر خبری ندارند. کلاینت بازدید کننده را به ساختار ابجکتی معرفی میکند و میگوید قبول کن. از سمت visitor به سمت المان DIP نقض میشود و دقت کن ساختار ابجکتی اون بازدید کننده را به المان میفرستد. بازدید کننده اجازه میدهد عملیات جدید اضافه کنیم. بازدید کننده باعث میشود عملیات های مرتبط یکجا جمع بشوند در قالب تخصص و اونهایی که مربوط به پیمایش نیستند در المان ها مینشینند و اونهایی که مربوط به پیمایش هستند و در ذات المان نیستند میروند در بازدید کننده ها و باعث بالا رفتن cohesion میشود. البته باعث افزایش

وابستگی میشود و کپسول سازی هم نقض میشود ولی موجه هست. اگر نوع جدید المان استفاده کنیم واسط بازدید کننده تغییر میکند و تغییر جدید به تمام بازدید کننده ها میرسد. کپسول سازی هم نقض میشود چون بازدید کننده داخل المان ها را میبینند.

فصل ششم:

اصول شی گرایی:

GRASP: الگو هایی هستند که به کلاس ها مسئولیت تخصیص میدهند.

OCP: سیستم ها و کلاس ها را باید بشود گسترش داد و در مقابل تغییر بسته باشند یعنی گسترش بدهیم بدون تغییر محتویات فعلی. شی گرایی باعث شد کپسول سازی پدید بیاد و این آرزو محقق شده است. میگوید چطور به سیستم ها قابلیت جدید اضافه کنیم بدون تغییر قبلی. یک روش این هست که وابستگی بین کلاس ها در سطح انتزاع باشد نه concreate ها مستقیم با هم. یا همون واسط ها. DIP مکانیزم برقراری OCP هست.

LSP: اصل قابلیت جایگزینی یعنی اینکه بتوان نمونه های ساب کلاس ها را بتوان به عنوان نمونه فوق کلاس جایگزین بشوند اگر نتوانیم این کار را بکنیم یعنی IS A نقض شده است. اگر نقض شود میشود همون میراث مردود که از پدر به فرزند به ارث میرسد ولی فرزند انجام نمیدهد ولی پدر متعهد هست. یعنی به صورت مکرر برای رشد پدر باید فرزندان هم چک کنیم اگر برقرار نشود مشکل هست و تغییر منتشر میشود از سمت فرزندان به پدر وابستگی شدید وجود دارد توی ساختار توارثی وابستگی بسیار زیاد هست فرزند نباید رفتار پدر را محدود بکند فقط باید گسترش بدهد چون LSP نقض میشود. میشود پدر را گسترش داد. یعنی شرایط پاسخگویی پدر را محدود میکند. نقض LSP همیشه به معنی تهی کردن رفتار نیست قوی تر کردن پیش شرط و ضعیف تر کردن پس شرط هم جز همین نوع هست.

DIP: concreate مستقیم به هم وصل نشوند و واسط باشد و گرنه مشکل هست و روابط در انتزاع باشد. و کلاینت به جای دید مستقیم به concreate فقط واسط را ببیند و وابستگی کمتر هست. مثل bridge که ۲ طرف رشد میکردند ولی مشکل به وجود نمیومد و واسط عوض نمیشد.

ISP: اصل تفکیک واسط ها میگوید که به جای اینکه برای یک کلاس یا ساب سیستم یک واسط تک خیلی بزرگ non cohesive بسازید واسط های بیشتر با انسجام بیشتر بسازید مثل همان facade که یک واسط ساده را در اختیار کلاینت میگذاشت. واسه همین عملیات های بی ربط به همدیگر را نباید در یک facade بگذاری باید چند تا باشد هر facade باید تک منظوره باشد. هر جا واسط تعریف میکنی باید به انسجام توجه کنی اگر پیچیده شد بشکن.

CRP: میگوید که ترجیح delegation بر توارث هست که همان مثال دانشجو که کلی ساختار سلب بود و مثال آن استیت و استراتژی بود که delegation را جایگزین کردیم و در زمان اجرا رفتار دینامیک میشود. به جای توارث برای استفاده مجدد بریم به composition برای استفاده مجدد یعنی یک ابجکت داخل ابجکت اصلی تعریف میکنیم و کار را به آن میسپاریم و ساختار توارثی را کنار میگذاریم تا پویایی بالاتر برود. از توارث به عنوان استفاده مجدد استفاده نکن. رابطه داشتن همان رابطه کل به جز نیست یعنی has a همان رابطه aggregation نیست و has a جایگزین بهتری به is a نیست به هر has a هم نمیگویند aggregation.

PLK: میخواهیم دید transitive نداشته باشیم یعنی چی ؟ الف با بی را میبیند و بی سی را میبیند الف به بی درخواست یک کار میدهد ولی به جای اینکه بی رفتار را از سی بگیرد و جواب الف را بدهد خود سی را به الف میدهد و این درست نیست چون نقض کپسول سازی هست چون سی جز حالت بی هست و نقض کپسول سازی میکنیم یا بهش message chain هم گفته میشود. بعد دید الف هعی گسترش پیدا میکند. باید بازیابی کنیم و زنجیره قطع شود و رفتار سمت داده قرار بگیرد. دید غیر مانا هست مگر بزاری داخل attribute. چه دیدی ممنوع هست؟ یک ابجکت به عنوان return value از یک فراخوانی برگردد و دید ما به آن برقرار شود.

GRASP: الگو های نرم افزاری عمومی تخصیص مسئولیت هستند.

جلسه پانزدهم:

۹ تا الگو هستند همین گرسپ،

الگو اول information expert: رفتار را نزدیک پیاده سازی بگذارید یا عملیات را بگذار نزدیک داده و کار را بسپارید به متخصص پیاده سازی آن یعنی جایی که اونجا با اون کار میکند اونجا قرار بگیرد. انسجام بالا میدهد و اگر رفتار کنار داده نباشد وابستگی زیاد میشود چون هعی میخواهند با هم کار بکنند و وابستگی میاد پایین. یا مسئولیت را بسپاریم به کلاس متخصص خودش. و اگر رخ ندهد وابستگی زیاد میشود.

Creator میگوید چطور مسئول ساختن ابجکت و نمونه گیری توسط چه کسانی انتخاب شود و بهترین factory کدام هست. میخواهیم مسئول ساختن ابجکت را به یک کلاس بسپریم و اگر به درستی ایجاد نشود باعث وابستگی میشود و میخواهیم وابستگی بی مورد ایجاد نشود. Composition یعنی اجزا بدون کل از بین میروند ولی در aggregate اجزا بدون کل از بین نمیروند و aggregation بیشترین وابستگی را دارد.

Low coupling: یک جوری باید که تخصیص مسئولیت میدهیم کمترین وابستگی داشته باشد. همیشه جوری مسئولیت بده که حداقل وابستگی ممکن باشد. اگر وابستگی باشد تغییرات گسترش پیدا میکنند و بر اساس تغییر دیگران

دچار تغییر میشوند. کلاسی که با کلاس دیگر وابستگی دارد به تنهایی نمیتوانی بررسی کنی و سخت هست چون از بقیه سرویس میگیرد. استفاده مجدد آنها هم کم هست چون وقتی میخوان مورد استفاده قرار بگیرند باید از اونها هم که بهش وابسته هست کمک بگیریم.

High cohesion میگوید که بیشترین انسجام در تخصیص مسئولیت ها باشد. و کلاس های چند مسئول نداشته باشیم. cohesion پایین یعنی کار های بی ربط داخل یک کلاس انجام میدهند معمولا هم در کلاس های بزرگ هستند. انسجام پایین استفاده از آن سخت هست استفاده مجدد از آن سخت هست و نگهداری آن هم سخت هست چون چند کاره هست و انسجام پایین هست و مدام تغییر میکند و divertance change دارد یعنی به ازای هر چیزی تغییر میکند چه UI تغییر کند هم تغییر میکند. Message chain نمود وابستگی زیاد و god class نمود انسجام کم هست.

Controller میگوید مسئولیت آمدن event ها از UI هستند چطوری باید توزیع و تخصیص داده شود یعنی اولین جایی که باید آن را بپذیرد و زنجیره تعاملی را راه بندازد. اینها بعد از واسط هستند روی لایه بیزنس لاجیک هستند و کلاس باید باشد وقایع را از کاربر بگیرد و زنجیره تعاملی را راه بیندازد. مسئولیت دریافت و کلید زدن جواب را بسپریم به کلاسی که این ویژگی ها را دارد: کل سیستم یا دیوایس یا یک ساب سیستم را مدیریت میکند انگار یک facade هست و روی بیزنس لاجیک نشسته است. کلاس UI که گیرنده واقعه هستند مناسب نیستند چون اینها گیرنده هستند منظور ما بعد از کلاس هست. همه ایونت ها به یک فساد نمیروند اون فسادی که مسئولش هست کنترلر میشود چون فساد هست. ممکن هست یک ابجکت خاص را برای راه انداختن یک یوزکیس خاص مسئول کنیم این مناسب نیست اینکه یک ابجکت یا کلاس درگیر در پاسخگویی انتخاب کنیم بهتر هست جا این یوزکیس بیس. یوزکیس کنترلر خوب هست وقتی که ابجکت درگیر مسئول شود نه دروغی.

POLYMORPHISM که همان چند نخی هست تاکید میکند جوری مسئولیت بدهیم که چند نخی اعمال شود چون سود دارد. وقتی رفتاری داریم که بر اساس تایپ یا کلاس متغیر هستند از چند نخی استفاده کن.

Indirection یعنی چطوری غیر مستقیم ارتباطات را برقرار بکنیم چون وابستگی بیاد پایین و یا باعث ترجمه مثل adapter میشوند یا رفتار اضافه میکنند. یعنی یک ابجکت میانی میدهیم برای کم کردن وابستگی و افزایش انعطاف تا اون کار ها را مدیریت کند. دید دو تا ابجکت نباید بعد از واسط برقرار باشد و گرنه بی معنی هست.

Pure fabrication میگوید که ابجکت هایی بسازیم که زاییده ذهن هستند مثل wrapper ها که میخواهیم قلمرو را کنترل کنیم برای افزایش انعطاف بازدید کننده هم جز همین ها هست. یک مجموعه از کار ها که انسجام بالا دارند به یک ابجکت مصنوعی که یعنی نمودی در قلمرو مسئله ندارد میدهیم برای آسان تر کردن. برای افزایش استفاده مجدد و کم کردن وابستگی. تو تحلیل این کار را نمیکنیم تو طراحی میکنیم .

Protected variation: یعنی اینکه چیکار کنیم بخش سیستمی که متغیر هست و تغییر پذیر هست چطوری روی بقیه سیستم تاثیر نگذارند چون اون ها پایدار نیستند و باعث عدم پایداری بقیه قسمت ها میشوند. ممکن هست قابل پیشبینی باشند یا نباشند و کاملا امکان پذیر هست برای این شرایط باید یک واسط تعریف کنیم و مسئولیت ها را به آن واسط بدهیم. یا facade یا ابجکت روی اون قسمت ناپایدار در قبال کلاینت بیرونی و وابستگی میاد پایین.

جلسه شانزدهم:

DBC: برای بالا بردن قابلیت اطمینان هست مهم ترین عنصری که استفاده میشود مفهوم assertion هست همون عبارت های ترو فالس هستند و در هر نقطه نشان میدهند حالت درست چی هست که باید ترو شود تا سیستم درست شود. ۳ تا assertion داریم ۲ تا متد یکی کلاس. برا متد پیش شرط و پسا شرط برای هر متد یک سری پیش شرط باید ترو باشند تا بتوان اون متد رو اجرا کرد و خروجی گرفت پسا شرط شرایطی هست که بعد از اجرا متد باید برقرار کند تا ترو شود بعد از اجرای این متد. Class invariant: یک شرط برای کل کلاس تعریف میشود و شرایطی را بیان میکند برای تمام ابجکت اون کلاس و گرنه ابجکت اون کلاس محسوب نمیشوند معمولا بیزنس رول ها را در قالب این کلاس ها تعریف میکنیم غیر از مواردی که دارد یک متد را اجرا میکند تا بعد از اتمام متد. Contract: دقیقا همین assertion هایی هستند بین ابجکت ها و کلاس ها و سرور باید پسا شرط را مهیا کند به شرطی که کلاینت پیش شرط را تضمین کند و نقض نکند و کل سیستم به کلاینت بیرونی سرویس میدهد. در توارث به فرزندان پیش شرط و پسا شرط و invariant ها به ارث میرسد اینها میتوانند قوی تر بشوند ولی نمیتوانند ضعیف تر بشوند چون ضعیف بشود فرزند صدق میکند ولی پدر نه و رابطه بهم میخورد چون پدر سخت تر بوده است و رابطه is a از بین میرود. پیش شرط میتواند ضعیف تر بشود ولی اگر قوی تر شود مشکل هست چون پدر آسان تر هست برای سرویس دهی ولی فرزند سخت گیر تر هست و نمیتواند جای پدر قرار بگیرد. رفتار در فرزند باید گسترش پیدا کند نه محدود شود برای زیر کلاس گیری. میتواند رفتاری که پدر قبول نکند فرزند قبول کند. در پسا شرط فرزند میتواند قوی تر بکند بیشتر از پدر یعنی بیشتر از پدر سرویس دهد و کار بکند. Is a نقض شود DBC نقض میشود.