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 2 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 3 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 2 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. 2 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 4 نوع دارد 1. Corrective یعنی یک جایی از کد خراب میشود تصحیح میکنیم. 2. Perfective یعنی وقتی feature جدید اضافه میشود ما تکمیل میکنیم کد را 3. Adaptive یعنی تطبیقی یعنی تغییری در زیر ساخت یا پلتفرم ایجاد میشود و ما باید نرم افزار را تطبیق بدهیم. 4. 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 2 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 2 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 2 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 2 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 2 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 طریق log 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 2 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 3 وضعیت دارد که میگیم رابطه کل به جز دارند 1. وقتی ابجکت های یک سمت تشکیل دهنده ابجکت های سمت دیگر باشند که بهش assembly گفته میشود. 2. 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. 3. 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 3 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 1. 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 2. 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 2 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 3 تا ستون دارد UI BUSINESS LOGIC و data. این انتشار تغییرات دارد بین این 3 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.