به نام خدا

تمرین دوم

گروه ۵

سوال اول

قابلیت ردیابی چیست؟

اگر علت چیزی را ندانیم، احتمالا نمی‌توانیم به چیستی آن پی ببریم؛ به همین دلیل ممکن است در بعضی مواقع قطعه‌کدها و طراحی‌های پیچیده و حتی ساده را بی‌معنا بدانیم.
منظور از قابلیت ردیابی نیازمندی‌ها در مهندسی نرم‌افزار، این است که بدانیم هر کدام از طراحی‌ها یا کدهای زده شده، مرتبط با کدام یک از نیازمندی‌های پروژه هستند. با این کار، می‌توانیم قسمت‌های به ظاهر بی‌معنای پروژه را دنبال کنیم تا به نیازمندیِ پسِ آن‌ها برسیم؛ در این صورت معنای آن‌ها را می‌فهمیم.

چه مزایایی دارد؟

امکان ردیابی نیازمندی‌ها، باعث فهم بهتر ارتباط گام‌های مختلف در توسعه‌ی پروژه می‌شود؛ به این معنی که اگر مثلا در مرحله‌ی طراحی داریم کاری انجام می‌دهیم، چرایی آن را بفهمیم و بدانیم که چرا داریم فلان چیز را طراحی می‌کنیم (چون مثلا مشتری از ما فلان مورد را خواسته و فلان مورد جزو نیازمندی‌هاست)، در مثالی دیگر و در زمان تست محصول، می‌توانیم بفهمیم آیا همه‌ی نیازمندی‌های پروژه مرتفع شده‌اند و از آن طرف، فقط نیازمندی‌های پروژه برطرف شده باشند.

خوبی دیگر قابلیت ردیابی، این است که اگر در قسمتی از پروژه تغییری ایجاد کنیم، می‌توانیم سایر قسمت‌هایی را که تحت تاثیر قرار گرفته‌اند، پیدا کنیم و به تبع، تغییرات لازم برای انطباق سایر بخش‌ها را در آن‌ها ایجاد کنیم.

ماتریس ردیابی چیست؟

ماتریس ردیابی (RTM) ابزاری است که به ما در مدیریت و بررسی وضعیت پیشرفت نیازمندی‌ها کمک می‌کند. این ماتریس باید همگام با پیشرفت پروژه به‌روزرسانی شود تا از به‌روزبودن همیشگی آن مطمئن باشیم؛ چرا که این ماتریس ابزار مهمی در جهت مدیریت پروژه است.

مثال:

برای مثال یک پروژه‌ی سامانه‌ی بانکی را در نظر بگیرید. این پروژه به دو نیازمندیِ «ورود» و «صفحه‌ی فرود یا landing» احتیاج دارد.

در ستون اول از ماتریس نیازمندی‌های زیر، نام این دو نیازمندی نوشته شده و در سایر ستون‌هایش، سایر اطلاعات مرتبط با نیازمندی از جمله هدف از این نیازمندی در پروژه، شخص یا دپارتمانی که این نیازمندی را مطرح کرده، آیدی تست‌هایی که برای اطمینان از این عملکرد درست نیازمندی نوشته شده، مسائل و مشکلات پیرامون نیازمندی، مرحله‌ی انجام نیازمندی و ... آورده شده است.

Requirements description	Business need/justification	Project objective	Requested by	Department	Testing	Deviation
Login page	Clients need access to accounts	Create minimum viable product	John D.	Content	1001	None
Landing page	Starting point for clients	Create minimum viable product	Christine M.	Content	1003	'Latest News' sidebar not updating

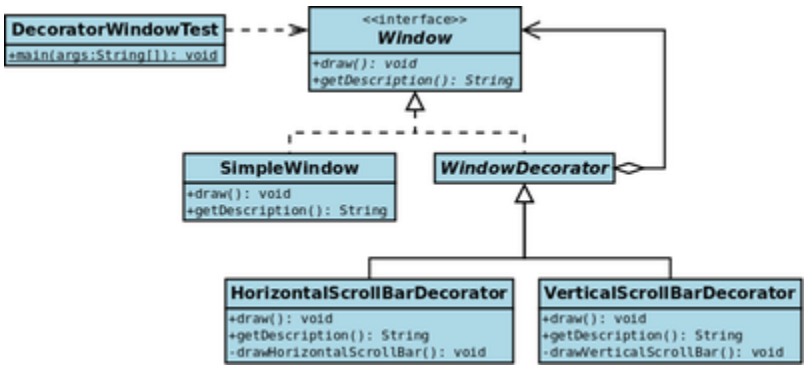
منابع

- https://www.sodiuswillert.com/en/blog/what-is-traceability-in-software-engineering
- https://www.wrike.com/blog/what-is-requirements-traceability-matrix/#:~:text=A%20traceability%20matrix%20is%20a,were%20defined%20for%20any%20requirement.

سوال دوم

مثال اول از الگوهای طراحی که به آن می‌پردازیم decorator pattern است. فرض کنید می‌خواهید یک آبجکت پیتزا درست کنید و این پیتزا می‌تواند انواع تاپینگ‌های مختلف مثل قارچ، زیتون، پنیر اضافه و ... را داشته باشد. Decorator pattern به ما این اجازه را می‌دهد تا بدون نیاز به ایجاد زیرکلاس‌های مختلف به ازای نوع تاپینگ‌های مختلف، بتوانیم پیتزاهای متنوعی بسازیم. به علاوه، این کار را نیز در زمان اجرا می‌توان انجام داد. ممکن است در حالت عادی فردی کلاس «پیتزا با قارچ و بدون پنیر اضافه» «پیتزا با قارچ و با پنیر اضافه»، «پیتزا بدون قارچ و پنیر اضافه» و ... را اضافه کند. حال آن‌که در این روش به ازای کلاس پیتزا یک decorator می‌سازیم و این decorator می‌تواند زیر کلاس‌های قارچ، پنیر اضافه، زیتون و ... را داشته باشد. سپس با صدا زدن متدی از این زیر کلاس‌ها شئ پیتزای خود را با استفاده از این attribute های جدید عملاً decorate می‌کنیم. این باعث می‌شود هم به صورت داینامیک بتوانیم attribute های جدید در طول اجرا اضافه کنیم و هم باعث می‌شود لازم نباشد تعداد بسیار زیادی زیر کلاس به ازای یک کلاس خاص ایجاد کنیم.

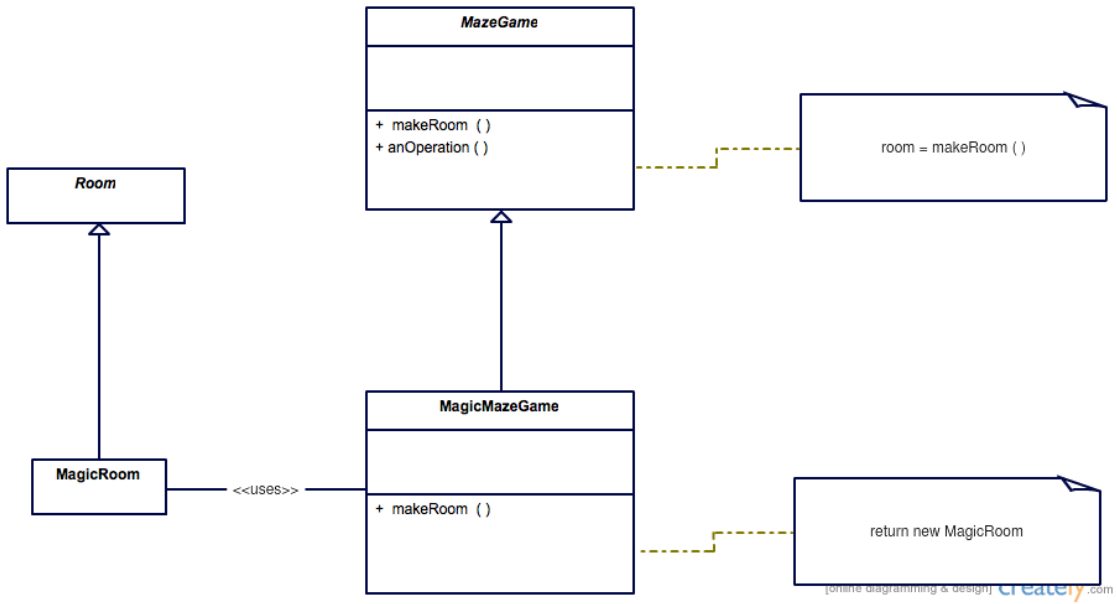
نمونه دیگر decorator در استفاده از این کلاس‌ها برای یک window است. هر windowمی‌تواند خصوصیات زیادی را داشته‌باشد. برای مثال می‌تواند در پایین صفحه یک اسکرول‌بار داشته باشد، در سمت راست یا چپ صفحه اسکرول بار داشته باشد، دورش قاب داشته باشد یا نداشته باشد و ... اگر بخواهیم به ازای هر حالت متفاوت یک کلاس متفاوت تعریف کنیم، این کار از نظر عملی شدنی نیست. لذا مطابق دیاگرام UML زیر عمل می‌کنیم و یک کلاس decorator به ازای کلاس windowمی‌سازیم و این attribute ها را با صدا زدن draw() در صورت نیاز به این window اضافه می‌کنیم.



مثال بعدی Command Pattern است که برای کم کردن کاپلینگ در برنامه طراحی‌شده‌است. در این متد کلاینت به جای اینکه به صورت مستقیم یک action از طریق receiver صدا زده نمی‌شود. بلکه Invoker مستقل از پیاده‌سازی یک آبجکت Command را به همراه receiver و پارامترهای ورودی صدا می‌زند و متد execute در

کلاس Command وظیفه انتقال دادن پارامترها و صدا زدن تابع action را برعهده دارد. با توجه به این جداسازی در صورت بروز خطا به راحتی می‌دانیم که باید در داخل تابع execute از کلاس Command به دنبال حل مشکل بگردیم.
برای مثال روی دکمه‌های GUI می‌توانیم از نمونه آبجکت‌های Command استفاده کنیم. برای مثال در Swing کلاس Action یک نوع Command آبجکت است که هنگامی که از سوی کلاینت دکمه فشار داده می‌شود، این کلاس متد execute اش را روی receiver خاص خود اعمال می‌کند و پارامترهای مربوطه را به آن انتقال می‌دهد.

مثال آخر Factory Method Pattern است که در آن یک کلاس Creator متد ابسترکت make() دارد که در آن آبجکتی ساخته می‌شود. توجه کنید از آنجایی که ممکن است آبجکت به انواع متفاوتی ساخته‌شود، زیرکلاس‌های مختلف Creator می‌توانند این تابع make را پیاده‌سازی کنند تا بتوانیم هدف‌های متفاوتی از کلاس make بگیریم.
فرض کنید می‌خواهیم یک بازی ماز درست کنیم و در این بازی دو نوع اتاق داریم، در نوع اتاق اول همیشه از یک خانه به خانه‌های مجاور می‌توانیم برویم ولی در نوع دوم می‌توانیم از بخش‌هایی از اتاق به بخش‌های دیگر teleport کنیم. مطابق شکل زیر می‌توانیم کلاس MazeGame را بسازیم که تابع makeroom در آن یک اتاق معمولی تولید می‌کند و در بقیه متدها room = makeroom() است. اگر بخواهیم به چنین کدی اتاق جادویی را نیز اضافه کنیم کافیست یک زیر کلاس برای MazeGame بسازیم و تابع makeroom آن‌را override کنیم و با این تغییر ساده می‌توانیم به راحتی یک اتاق جادویی با همان کد قبلی تولید کنیم. توجه کنید در این نمونه Creator های ما MazeGame و MagicMazeGame هستند و تابع make() ما همان makeroom() است که اتاق‌ها مختلفی می‌سازد.



سوال سوم

Information Hiding

پنهان سازی اطلاعات در مهندسی نرم افزار روشی برای جلوگیری از تغییرات غیر قابل برگشت و ناخواسته در طراحی و پیاده سازی قسمتی از برنامه، هنگام تغییر قسمت های دیگر برنامه می باشد برای مثال اگر در طراحی، دسترسی به متغیرهای کلاس ها را از بقیه کلاس ها مخفی کنیم احتمال تغییرات ناخواسته روی کلاس مخفی شده کمتر میشود.

در طراحی شی گرا روش هایی برای برای پنهان سازی داده ها وجود دارد. فرض کنید یک کلاس ماشین داریم. کلاس های بیرونی تنها به متد روشن کردن، دسترسی دارند و کاری به نحوه ی اجرا و متغیر های دیگر کلاس ندارند. حال اگر بخواهیم نحوه روشن کردن ماشین را تغییر دهیم، این تغییر تاثیری روی طراحی کلاس های بیرونی ندارد و همچنین احتمال تغییرات ناخواسته روی متغیرهای دیگر کلاس توسط کلاس های بیرونی کاهش میابد. اعمال این نوع پنهان سازی با استفاده از private کردن متغیر ها امکان پذیر است.

به این صورت از مزیت های این کار میتوان به موارد زیر اشاره کرد

- جلوگیری از تغییر و یا تخریب داده ها با پنهان کردن آن ها از عموم
- بالا بردن cohesion با خارج کردن دسترسی به اطلاعات غیر مرتبط با کلاس ها
- جلوگیری از سو استفاده و هک توسط مهاجمان

سوال چهارم

در تحلیل تمرکز ما صرفا بر روی use-case ها و استخراج نیازمندی‌هاست، لذا انواع کلاس‌هاییکه مرتبط با راه‌حل می‌شوند در این حالت ارائه نخواهند شد اما در نقطه مقابل بسیاری از کلاس‌های لایه طراحی مستقیما به حل مسئله و ارتباط کلاس‌های مختلف می‌پردازند.
فرض کنید از یک معماری Model View Controller - MVC برای حل مسئله استفاده می‌کنید. در این حالت، در مرحله طراحی صرفا به کلاس‌های مرتبط با Model می‌پردازید تا بتوانید نیازمندی‌های بیزینس یا کار خود را به صورت روشن دریابید. اما در هنگام طراحی روی Controller ها و C تمرکز می‌کنیم. از این‌رو کلاس‌هایی مثل Decorator، Command، Controller و انواع سینگلتون‌ها مختص طراحی هستند و در مرحله تحلیل نباید مطرح شوند.

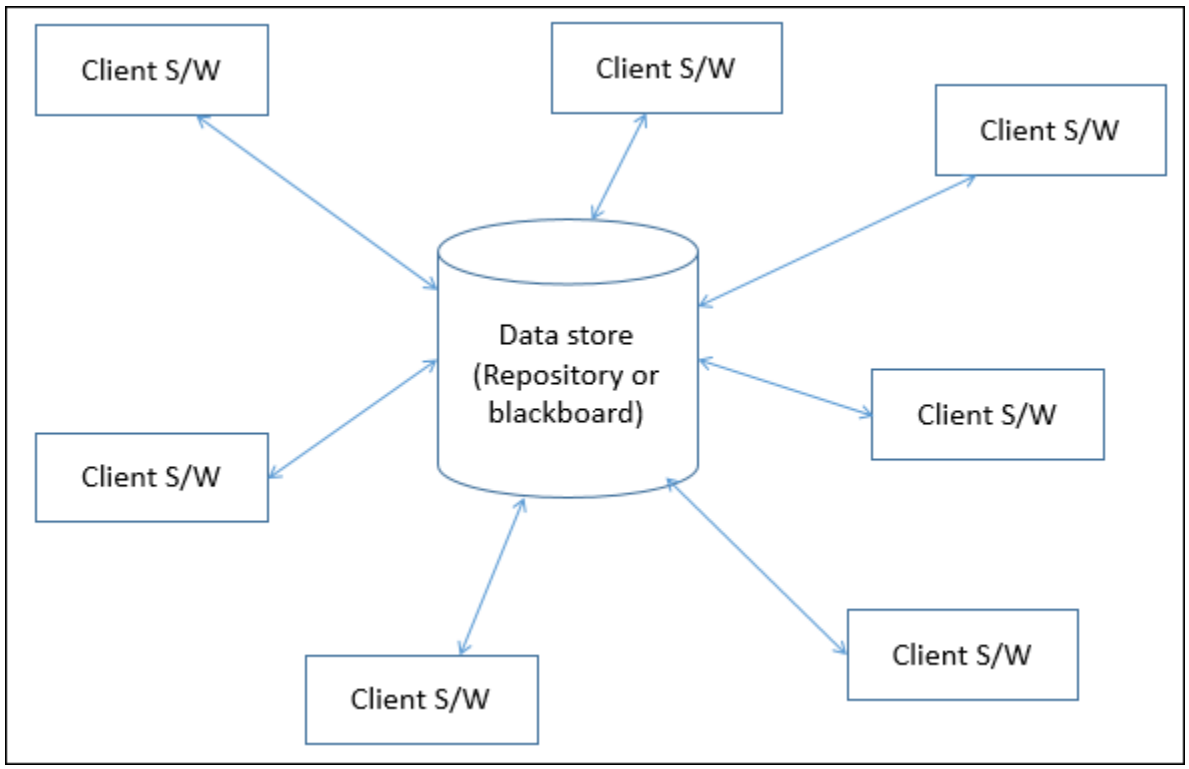
از طرفی مدل‌های ارائه شده در مرحله تحلیل پروتوتایپی هستند برای شروع. در نتیجه عموما کلاس‌های ارائه شده در زمان تحلیل ناکامل هستند و عاری از جزئیات هستند. در حالی که ممکن است متدهای جدیدی در مرحله طراحی به کلاس اضافه شود.

سوال پنجم

Data-centered architectures

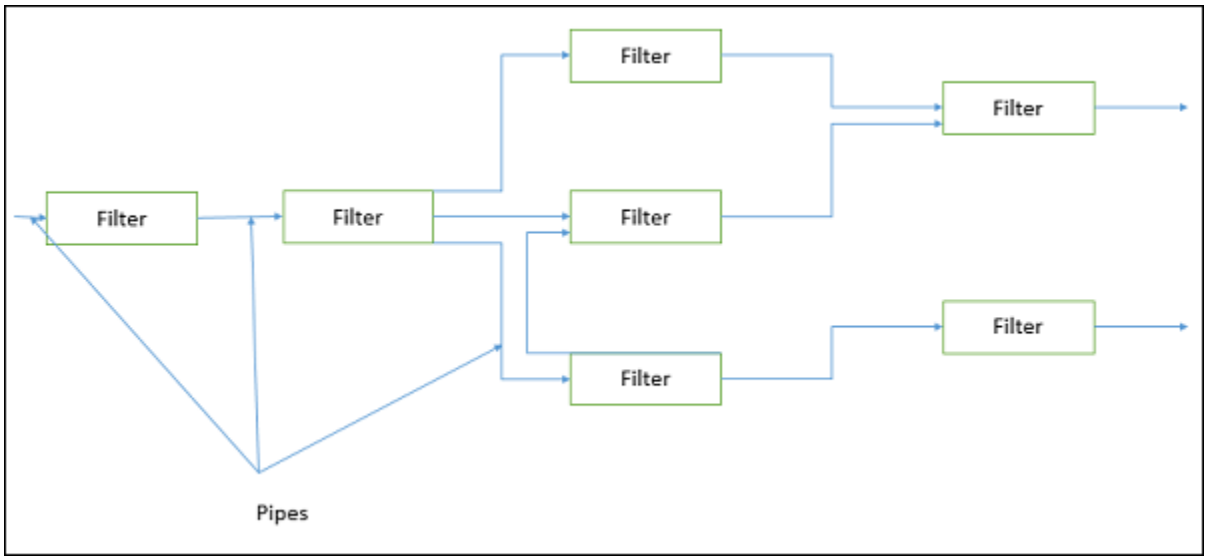
در معماری data-centered، داده در مرکز قرار دارد و مولفه‌های نرم‌افزاری هر کدام در یک گوشه با مرکز داده (data store) ارتباط برقرار میکنند و داده‌های مرکز داده را تغییر میدهند. ارتباط مستقیم بین مولفه‌های کناری با هم وجود ندارد. دو نوع Repository و Blackboard برای این معماری میتوان در نظر گرفت. در نوع Repository، مولفه‌های تغییر دهنده داده‌ها در مرکز داده هستند. در مولفه‌ها (data accessors) تغییرات را آغاز (trigger)میکنند. DBMSها، سیستم‌های کنترل نسخه (مانند گیت) و سیستم‌های توسعه نرم‌افزار بر روی ابر (cloud based development systems) از این نوع هستند.

در نوع Blackboard، مولفه‌ها منفعل هستند و مرکز داده کنترل جریان کار را به عهده میگیرد. در صورت بروز یک رخداد، مرکز داده است که به مولفه‌ها پیام ارسال میکند. در واقع مولفه‌ها تابع مرکز داده هستند و نه برعکس. ارتباط مولفه‌ها در این معماری، با مکانیزم تخته سیاه انجام میشود (blackboard mechanism).

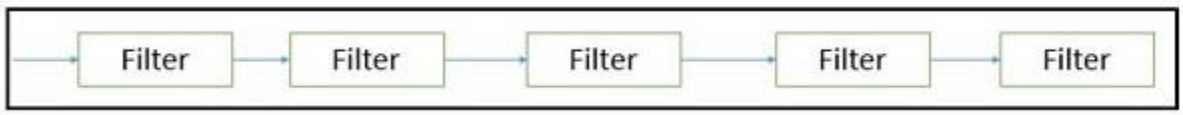


Data flow architectures

در معماری‌های جریان داده، سیستم از تعدادی مولفه کوچک تشکیل شده که این مولفه‌ها پشت سر هم قرار میگیرند و هر کدام بر روی داده ورودی خود تغییراتی را اعمال میکنند (مانند فیلتر کردن یا تغییر شکل و mapping و انجام عملیات و ...) و خروجی میدهند. خروجی یک مولفه میتواند ورودی یک مولفه دیگر باشد. از کنار هم قرار گرفتن این مولفه‌ها، خروجی نهایی سیستم به‌دست می‌آید. تصویر زیر یک نما از معماری جریان داده است.



خط فرمان سیستم‌عامل لینوکس یک مثال خوب از این نوع معماری است. برای به‌دست آوردن یک خروجی خاص، میتوانیم از تعدادی برنامه استفاده کنیم (command line utilities) و با پشت سر هم قرار دادن آن‌ها در یک خط لوله (pipeline) میتوانیم خروجی مورد نظر را بگیریم. مانند تصویر زیر:



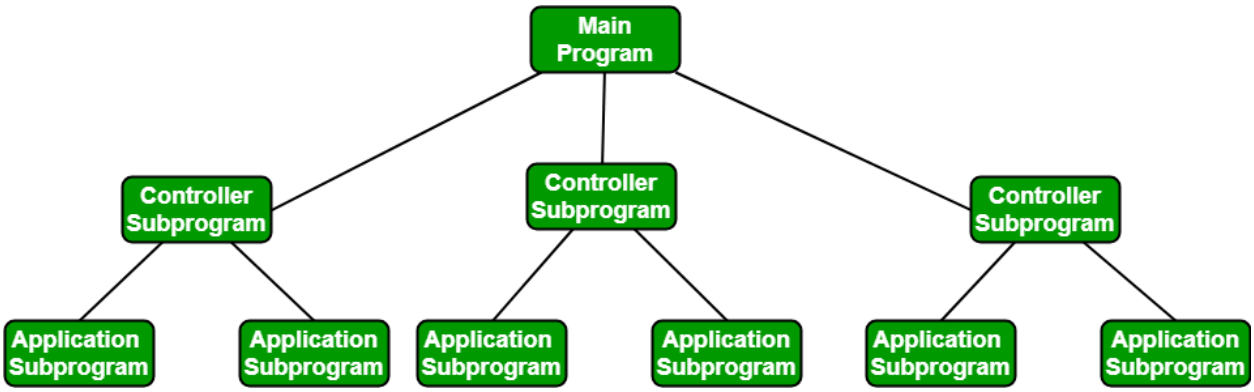
نکته قابل توجه در این معماری، قابلیت جابه‌جایی هر کدام از مولفه‌هاست. کافی است که واسط‌های مولفه‌ها (interfaces) مشخص باشند. در این صورت میتوان به جای یک مولفه، یک مولفه جایگزین دیگر که با همان واسط کار میکند قرار دهیم.

Call and return architectures

در این نوع معماری، نرم‌افزار از تعدادی مولفه تشکیل شده که مولفه‌های نرم افزاری با فراخوانی یکدیگر خروجی مورد نظر را تولید میکنند. دو زیر نوع در این نوع معماری وجود دارد.

نوع اول معماری Remote Procedure Call است. در این معماری، مولفه‌ها میتوانند به صورت جداگانه از هم قرار بگیرند (پردازه‌های متفاوت) یا اصلا در ماشین‌های جدا قرار بگیرند. از چارچوب (gRPC) میتواند برای پیاده‌سازی چنین معماری استفاده کرد. در واقع پیاده‌سازی معماری میکروسرویس با استفاده از این استایل شایع است.

نوع دوم معماری main program or subprogram است. در این معماری، برنامه اصلی از تعدادی زیربرنامه تشکیل شده. خود زیربرنامه‌ها هم از زیربرنامه‌های دیگری تشکیل شده اند و به این صورت یک مولفه با فراخوانی تعدادی مولفه دیگر خروجی را تولید میکند.



Object-oriented architectures

در این نوع معماری، نگاه Object Oriented در سیستم وجود دارد. یعنی سیستم به تعدادی شئ تقسیم میشوند و خروجی مطلوب سیستم از تعامل اشیاء و جابه‌جایی پیغام بین این اشیاء (فراخوانی یا استفاده از rpc) به‌دست می‌آید. تمام سیستم‌هایی که با نگاه Object Oriented ساخته می‌شوند، مثالی از این نوع معماری هستند.

Layered architectures

در این نوع معماری، سیستم از چند مولفه تشکیل شده که به آن‌ها لایه نیز میگوییم. لایه ها از بالا به پایین قرار گرفته‌اند. این قاعده وجود دارد که هر لایه فقط و فقط میتواند با لایه زیرین یا زبرین خود ارتباط کند. چنین معماری در برنامه‌های تحت وب (web applications) بسیار رایج است. لایه واسط (مانند ui)، لایه application، و لایه data access، از لایه‌های معمول یک بکند (backend) سه لایه برای یک برنامه تحت وب است.

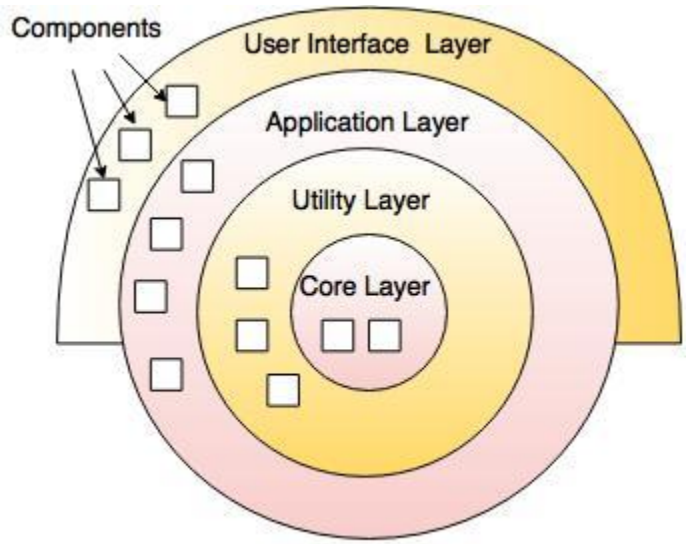


Fig.- Layered Architecture