سبک‌ها و الگوهای متداول معماری نرم‌افزار در یک نگاه

علی کریمی
استادیار دانشگاه جامع امام حسین (ع)
وحید ستوده قره‌باغ
دانشجوی دکتری دانشگاه جامع امام حسین (ع)
حسین خلیلی
دانشجوی کارشناسی ارشد دانشگاه جامع امام حسین (ع)

چکیده

برای توسعه‌ی یک نرم‌افزار، گام‌های مختلفی باید برنامه‌ریزی و اجرا شود که از این گام‌ها با عنوان فرایند توسعه‌ی نرم‌افزار یاد می‌شود. تحلیل‌گر، طراح، معمار و سایر افرادی که در تیم توسعه‌ی نرم‌افزار نقش دارند، در صورتی‌که از همان ابتدا به خصوصیات کیفی نرم‌افزار همچون مقیاس‌پذیری، تجمیع‌پذیری، قابلیت تغییر، قابلیت اطمینان و غیره توجه نکنند، نرم‌افزار تولیدشده فاقد کیفیت لازم خواهد بود؛ حتی اگر تمامی نیازمندی‌های وظیفه‌مندی را نیز پیاده‌سازی نموده باشد. معماری نرم‌افزار یکی از سازوکارهای حیاتی و اثربخش سازماندهی سامانه‌های نرم‌افزاری است. در این فرایند، الگوهای معماری به عنوان پایه‌ای برای طراحی است تا خصوصیات کیفی ذکر شده را تضمین کند. این الگوها از طریق بهینه‌سازی فرایند توسعه و نگهداری، بهبود کیفیت و عملکرد، افزایش قابلیت اطمینان و پایداری سیستم‌ها، افزایش رضایتمندی کاربران و کارایی کسب‌وکارها را به همراه دارد. شناخت ویژگی‌های موجود در آن‌ها، انتخاب الگو(ها)ی مناسب با ویژگی‌های خاص نرم‌افزار درحال‌توسعه را تسهیل نموده و منجربه استفاده صحیح از آن‌ها خواهد شد. با توجه به اهمیت این موضوع در فرایند توسعه نرم‌افزار، در این مقاله ۱۳ مورد از الگوهای متداول معماری نرم‌افزار را از حیث ساختار، ویژگی‌ها، مزایا و معایب بررسی نموده و نحوه‌ی به‌کارگیری آن‌ها را با هدف بهبود کیفیت و عملکرد نرم‌افزار، به‌طور اجمالی مورد بررسی قرار خواهیم داد.

واژگان کلیدی: معماری نرم‌افزار، الگوهای معماری نرم‌افزار، خصوصیات کیفی، تضمین کیفیت

۱- مقدمه

با گسترش روزافزون دامنه درخواست‌های کاربران رایانه‌ها و به دنبال آن اندازه سیستم‌های نرم‌افزاری، دیگر روش‌ها و سبک‌های کلاسیک تولید نرم‌افزار پاسخگوی این نیازمندی‌ها نیست. انتخاب یا ایجاد ساختمان داده و یا استفاده از الگوریتم‌های کارا، منجر به طراحی یک نرم‌افزار موفق نمی‌شود. حجم نرم‌افزارهای تجاری در سالیان اخیر، دلیل استفاده از روش‌های مبتنی بر مولفه‌ها، غلبه بر پیچیدگی‌های این حجم بالا است (Olukunle & Oyerinde, 2022).

آیا تابه‌حال به نحوه ساخت یک سیستم نرم‌افزاری با کارایی بالا فکر کرده‌اید؟ در سال‌های اخیر، کاربرد نرم‌افزار بسیار فراگیر شده است. به‌زودی هیچ دستگاه، ماشین یا امکاناتی وجود نخواهد داشت که کنترل آن توسط نرم‌افزار یا قطعات نرم‌افزاری اجرا نشود؛ بنابراین، نرم‌افزار برای عملکرد صحیح دستگاه‌ها و صنایع بسیار مهم است. به همین ترتیب، عملکرد روان یک شرکت یا سازمان تا حد زیادی به قابلیت اطمینان سیستم‌های نرم‌افزاری مورد استفاده برای پشتیبانی از فرایندهای تجاری و وظایف خاص بستگی دارد. در هر دو سیستم نرم‌افزاری تعبیه‌شده و تجاری، کیفیت به مهم‌ترین عامل در تعیین موفقیت تبدیل شده است. بسیاری از شرکت‌ها این وابستگی به نرم‌افزار را تشخیص داده‌اند و برای بهبود کیفیت سیستم‌های نرم‌افزاری و توسعه فرایندهای مهندسی نرم‌افزار خود تلاش می‌کنند (Spillner & Linz, 2021).

اگر بخواهیم از معماری نرم‌افزار تعریفی ساده داشته باشیم، می‌توان گفت: «معماری یک سیستم نرم‌افزاری یا محاسباتی، ساختار یا ساختارهای آن سیستم است که خصوصیات قابل رؤیت از بیرون مؤلفه‌ها و ارتباطات بین آنها را نشان می‌دهد» (Bass et al., 2021).

علاوه بر کاهش ریسک، طراحی مناسب، بهره‌وری توسعه‌دهنده را افزایش می‌دهد (به‌عنوان مثال، بهبود قابلیت نگهداری، قابلیت حمل و انعطاف‌پذیری)؛ بنابراین، برای انعکاس بهتر چالش‌های طراحی نرم‌افزار، تیم‌های توسعه تغییرات طراحی معماری (احتمالاً با شعار «پیشگیری بهتر از درمان است») را به طور منظم یا پس از تکمیل نقاط عطف یا انتشار بررسی می‌کنند. گفته می‌شود، نگرانی‌های طراحی نرم‌افزار از طریق معماری نرم‌افزار گرفته می‌شود (Mondal et al., 2022).

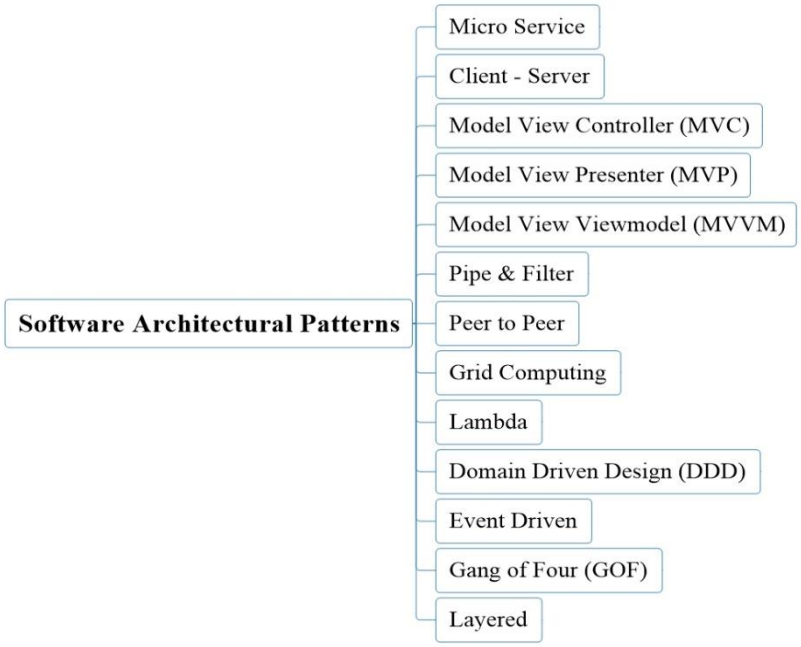
معماری نرم‌افزار به دلایل زیادی در هموارکردن راه برای موفقیت نرم‌افزار مهم است. این ساختاری است که رضایت از ویژگی‌های سیستمی (مانند عملکرد، امنیت، دردسترس‌بودن، قابلیت تغییر و غیره) را تسهیل می‌کند (Kassab et al., 2018).

معماران ممکن است با مسائل تکراری در طراحی معماری نرم‌افزارهای مختلف مواجه شوند. برای صرفه‌جویی در هزینه‌های هنگفت و کاهش خطرات، تصمیمات معماری نرم‌افزار می‌تواند بر مجموعه‌ای از الگوهای اصطلاحی که معمولاً سبک‌ها یا الگوهای معماری نامیده می‌شوند، تکیه کنند (Kassab et al., 2012).

یک الگوی معماری نرم‌افزار خانواده‌ای از سیستم‌ها را بر اساس الگوی سازماندهی و رفتار ساختاری تعریف می‌کند. به طور خاص، یک الگوی معماری، واژگان اجزا و رابطهایی را که می‌توان در نمونه‌هایی از آن الگو استفاده کرد، به همراه مجموعه‌ای از محدودیت‌ها در مورد نحوه ترکیب آنها، تعیین می‌کند. الگوهای معماری با استفاده از چارچوب‌های مختلف توصیف شده‌اند. یک چارچوب مشترک برای توصیف الگوها شامل نام، مشکل، ساختار و پویایی راه‌حل، و پیامدهای استفاده از یک الگو بر حسب مزایا و تعهدات آن است (Kassab et al., 2018). از این رو، انتخاب یک الگوی معماری از نظر کیفی با ویژگی‌هایی که نشان می‌دهد و دانش و تجربه معمار در مورد الگوها هدایت می‌شود (Bode & Riebisch, 2010).

باتوجه‌به مطالعات انجام شده در حوزه متداول‌ترین الگوهای معماری نرم‌افزار با بررسی مقالات پیشین، آرایه‌شناسی الگوها به‌صورت شکل ۱ ارائه می‌شود. در این شکل، ۱۳ الگوی موجود و متداول معماری در فرایند معماری نرم‌افزار جداگانه بررسی شده‌اند.

بخش باقی‌مانده از کار به شرح زیر تنظیم شده است. بخش دوم مفهوم معماری نرم‌افزار را مورد بحث قرار می‌دهد. در بخش سوم، مفهوم الگوی معماری نرم‌افزار بررسی و همچنین ماهیت و چگونگی هر یک از ۱۳ الگو، ارائه خواهد شد. بخش چهارم شامل مقایسه الگوهای مورد بحث است. بخش پنجم شامل نتیجه‌گیری است.



شکل ۱ – آرایه‌شناسی الگوهای رایج و متداول معماری نرم‌افزار (کریمی و همکاران ۱۴۰۲)

۲- معماری نرم‌افزار[1]

سالیان متمادی، طراحان نرم‌افزار سیستم را بر اساس نیازمندی‌های عملیاتی و فنی توسعه می‌دادند. مستندات نیازمندی‌ها به طراح داده می‌شد و او می‌باید طراحی قابل‌قبولی ارائه می‌داد. روش‌های جدید تولید نرم‌افزار، این مشکل را درک نموده و چرخه‌ای برای انتقال بازخورد اطلاعات و نظرات از طراح به تحلیل‌گر و برعکس، ارائه نمودند (Bass et al., 2021).

معماری نرم‌افزار یک نمایش سطح بالا است که ساختار اصلی و تعاملات اجزای داخلی یک سیستم و تعاملات بین سیستم و محیط آن را تعریف می‌کند. محققان و دست‌اندرکاران، معماری نرم‌افزار را به‌عنوان ساختار سازه، اجرای تصمیمات طراحی، تکامل و مکانیسم‌های اشتراک دانش یک سیستم تعریف می‌کنند (Mondal et al., 2022).

معماری نرم‌افزار، ابزار، فنون، ایده‌ها و روش‌هایی را با هدف پی‌ریزی ساختار پایه‌ی یک سیستم نرم‌افزاری ارائه می‌دهد (Hassan & Oussalah, 2018).

در توسعه نرم‌افزار مراحل مختلفی وجود دارد که توسعه‌دهندگان برای توسعه نرم‌افزار طی می‌کنند. از این مراحل، مرحله طراحی نقش زیادی در موفقیت سیستم دارد. طراحی معماری یک فرایند طراحی سطح بالا است که شامل شکستن سیستم بزرگ به عناصر کوچک‌تر، تعریف رابطه بین عناصر و مشخص‌کردن محیطی است که در آن کار می‌کنند. تصمیمات معماری هسته موفقیت

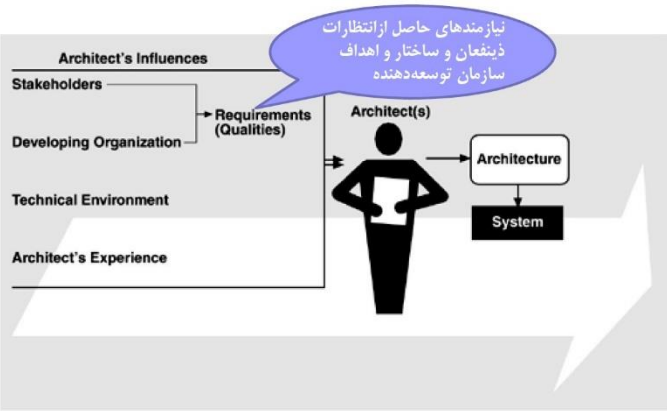
[1] Software Architecture

نرم‌افزار هستند. فرایند طراحی معماری را می‌توان از ابتدا انجام داد یا می‌توان با یک الگوی معماری انتخاب شده به آن کمک کرد (Aychew & Alemneh, 2022).

معماری نرم‌افزار به‌عنوان یک طرح اولیه برای فرایند توسعه یک سیستم عمل می‌کند. معماری نرم‌افزار می‌تواند یک سند طراحی برای حفظ کیفیت سیستم‌ها باشد. اجزای نرم‌افزار، عملکرد آنها و تعامل آنها با یکدیگر را شرح می‌دهد (Shreelekhya et al., 2016).

معماری نرم‌افزار حاصل مجموعه‌ای از تصمیمات فنی[1] و تصمیمات کاری[2] است. عوامل زیادی در معماری نرم‌افزار تأثیرگذار هستند و تحقق این عوامل، بسته به محیط اجرای معماری، تغییر خواهد کرد. اولین و مهم‌ترین عامل تأثیرگذار، نیازمندی‌های سیستم است که نیازمندی‌های سیستم شامل دو دسته «نیازمندی‌های وظیفه‌مندی[3]» و «نیازمندی‌های غیر وظیفه‌مندی[4] (یا خصوصیات کیفی[5])» است (Bass et al., 2021). تاثیر عوامل بر معمار در شکل ۲ نشان داده شده است.

محیط فنی، کسب‌وکار و فرهنگ سازمان توسعه‌دهنده، بر معماری نرم‌افزار تأثیر می‌گذارد. متقابلاً، معماری نرم‌افزار نیز محیط فنی، کسب‌وکار و فرهنگ سازمان تأثیرگذار است. چرخه تأثیرات از محیط به معماری و بالعکس را «چرخه کاری معماری[6]» گویند (Bass et al., 2021). چرخه کاری معماری در شکل ۳ نشان داده شده است.

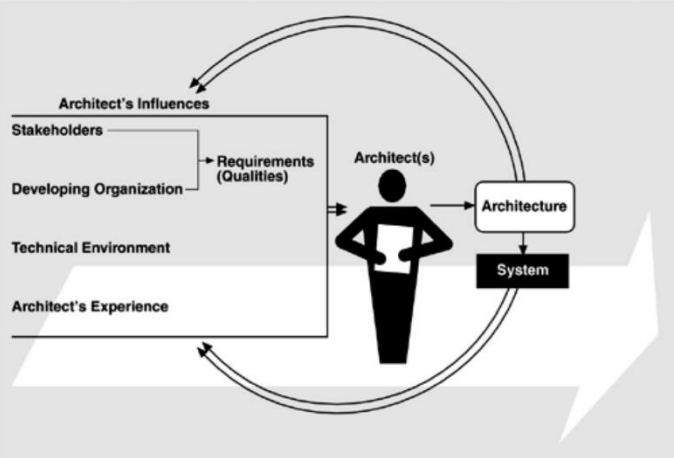


شکل ۲ – تاثیر عوامل بر معمار (Bass etal., 2012)

برخی بازخوردها مستقیم از

برخی دیگر از سیستم ساخته‌شده از

[1] Technical
[2] business
[3] Functional Requirements
[4] Non-Functional Requirements
[5] Quality Attributes
[6] The Architecture Business Cycle



شکل ۳ - چرخه کاری معماری (Bass etal., 2012)

۳- الگوهای معماری نرم‌افزار[1]

الگوی معماری مجموعه‌ای از تصمیم‌های طراحی است که مکرراً در عمل یافت می‌شود، دارای ویژگی‌های شناخته‌شده‌ای است که اجازه استفاده مجدد را می‌دهد و کلاسی از معماری‌ها را توصیف می‌کند (Aychew & Alemneh, 2022).

الگوی معماری یک طرح سازمانی ساختاری اثبات‌شده برای سیستم‌های نرم‌افزاری و توصیف مجموعه‌ای از زیرسیستم‌های از پیش‌تعریف‌شده و مسئولیت‌های آنهاست (Onarcan & Fu, 2018).

معمولاً با استفاده از مؤلفه‌ها، تعاملات و روابط متقابل و مفاهیم تعریف می‌شود (Jacob & Mani, 2018). بنابراین، برای انتخاب الگوهای معماری، ادبیات مختلف از ویژگی‌های متفاوتی استفاده می‌کند: خصوصیات کیفی (Bass et al., 2021; Harrison & Avgeriou, 2010)، نیازمندی‌های عملیاتی (Babar & Gorton, 2009) و محدودیت‌های (Velasco-Elizondo Sabry, 2015; et al., 2016) موجود در الگوهای معماری.

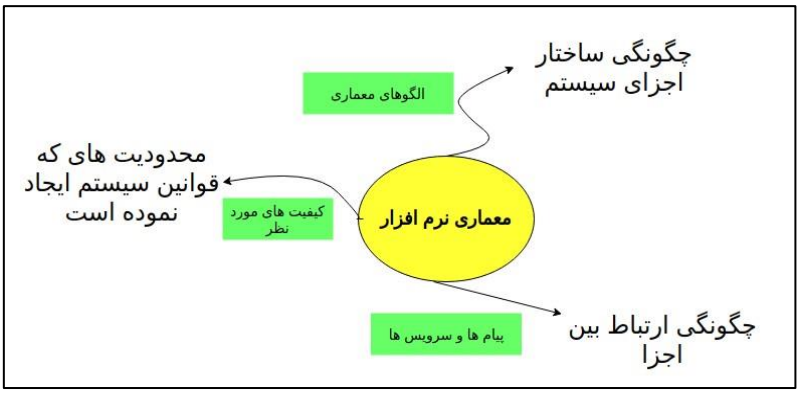
تشخیص زودهنگام الگوهای معماری برای موفقیت توسعه حیاتی است. ازآنجایی‌که محیط‌های مختلف توسعه دارای الگوهای معماری متفاوتی هستند، تشخیص اولیه الگوی می‌تواند به انتخاب محیط کار کمک کند (Harrison & Avgeriou, 2010). بنابراین، اگر ما یک طراحی معماری خوب بسازیم، موفقیت نرم‌افزار را تضمین می‌کنیم.

الگوهای معماری، توصیفی از عناصر و انواع ارتباطات به همراه مجموعه‌ای از محدودیت‌ها در مورد چگونگی استفاده از آنها است. الگوهای معماری، اجبارهای مفیدی روی معماری اعمال می‌کنند؛ اما خودشان معماری نیستند. انتخاب الگوی معماری، نخستین انتخاب مهم معمار در طراحی یک معماری است. پرکاربردترین جنبه‌های الگوها، توانایی نمایش خصوصیات کیفی معماری نرم‌افزار است که یکی از دلایل مهم انتخاب الگوی مشخص به‌جای الگوی تصادفی توسط معمار می‌باشد (Bass et al., 2021). بخش‌های معماری نرم‌افزار در شکل ۴ نشان داده شده است.

الگوها با ادغام فنون مختلف پیاده‌سازی شده برای بیان یا اندازه‌گیری ویژگی کیفی سیستم ساخته می‌شوند. ازآنجایی‌که سلول‌ها بلوک ساختمانی زندگی هستند، فنون، بلوک ساختمانی طراحی معماری هستند، زیرا مجموعه فنون مختلف پیاده‌سازی شده بر روی سیستم برای پرداختن به ویژگی‌های کیفی مختلف، الگوهای معماری را می‌سازد (Kassab & El-Boussaidi, 2013; Scott & Kazman, 2009). اما ویژگی‌های کیفیت، مشخصات یک موجودیت هستند که بر توانایی آن در ارضای نیازهای اعلام شده یا ضمنی تأثیر می‌گذارند.

[1] Software Architectural Patterns

اهمیت فنون معماری در این واقعیت نهفته است که آنها راه‌حل‌های ابتدایی هستند که الگوهای معماری به‌دست آمده در معماری نرم‌افزار را حفظ می‌کنند. در این راستا، بیشتر الگوهای معماری از چندین فن مختلف معماری تشکیل شده‌اند (از آنها ساخته شده‌اند) (Márquez et al., 2023).



شکل ۴ - بخش‌های مختلف معماری نرم‌افزار

الگوی طراحی[1] (یا فقط الگو) به معنای راه‌حل کلی قابل تکرار برای یک مسئله یا سناریو در مهندسی نرم‌افزار و الگوی طراحی معماری[2] (یا فقط الگوی معماری) برای اشاره به ترکیبی از الگوهای طراحی یا سایر ساختارهای سازمانی است که در یک معماری به خوبی تعریف شده یا قطعه‌ای از آن چیده شده‌اند (Ortiz Fuentes & Herranz Nieva, 2022). مجموعه مهم و مستندی از الگوهای طراحی و الگوهای معماری را می‌توان در (Gamma et al., 2015; Yu et al., 2017) و (Fowler, 2012) یافت.

به‌طورکلی، در الگوی معماری محدوده سطح بالای سیستم مشخص می‌شود، مانند ساختار اجزای سیستم و اما در الگوهای طراحی محدوده سطح پایین سیستم تعریف می‌شود، مانند اینکه یک جزء سیستم چگونه پیاده‌سازی می‌گردد و توجه داشته باشید در الگوی معماری نحوه پیاده‌سازی مشخص نمی‌شود. به‌عنوان مثال در الگوهای طراحی نحوه سرویس‌دهی به‌صورت توزیع شده را مشخص و یا لایه‌های سیستم تعیین می‌گردد.

یک الگوی معماری، معماری نیست؛ اما کماکان یک تصویر قابل استفاده از سیستم ارائه می‌کند. مسائل غیرقابل‌حل در یک الگو عبارت‌اند از (Bass et al., 2021):

- تعداد دقیق عناصر و ارتباطات بین آنها مشخص نیست. مثلاً در الگوی کلاینت - سرور، معمولاً تعداد کلاینت‌ها مشخص نیست.
- رفتار عناصر در کاربرد الگوها مشخص نیست. رفتار عناصر، بخشی از معماری است و نه الگوی معماری.

۳-۱- زبان الگو

به مجموعه‌ای از قواعد، الگوها و قوانین که برای ساخت و سازماندهی الگوهای جدید از روی الگوهای اولیه استفاده می‌شود، زبان الگو می‌گویند. در دنیای یک زبان الگو، الگوها نقش مجموعه لغات زبان را بازی می‌کنند، درحالی که قوانین، گرامر زبان را تشکیل می‌دهند و مشخص می‌سازند که یک الگوی جدید، به‌چه نحوی باید ساخته شود. همچنین در برخورد با یک الگوی جدید و

[1] Design Pattern
[2] Architectural Design Pattern

ناشناخته، این قوانین به ما کمک می‌کنند تا تشخیص دهیم که آیا این الگو، یک الگوی معتبر این زبان است یا خیر. به کمک این تعریف می‌توان خط مرزی میان الگو و سبک را ترسیم کرد.

۳-۲- الگوی طراحی در مقایسه با الگوی معماری

الگوی طراحی و الگوی معماری از آن جهت به هم شباهت دارند که هر دو با این فلسفه به وجود آمده‌اند که طراح و معمار را در انتخاب راه‌حل مناسب یاری دهند و به همین منظور هر کدام راه‌کارها و دستورالعمل‌هایی را تجویز می‌کنند. اما در سطح درشت‌دانگی با یکدیگر تفاوت دارند. الگوی طراحی معمولاً یک راه‌حل جزئی‌تر و دقیق‌تر برای مسئله مورد نظر پیشنهاد می‌کند، به‌عنوان مثال؛ اگر گفته شود در یک سیستم از الگوی طراحی پل[1] استفاده شده است، هر چند به‌صورت کلی، این دید حاصل می‌شود که پیاده‌سازی یک نوع داده‌ای انتزاعی در قالب یک کلاس واسط و یک کلاس پیاده‌ساز انجام شده است. همچنین، می‌دانیم که این کار برای جداسازی پیاده‌سازی از واسط صورت‌گرفته تا در صورت تغییر در پیاده‌سازی با ثابت نگاه‌داشتن کلاس، واسط برنامه‌های استفاده‌کننده، دستخوش تغییر نشده و تغییرات در سیستم منتشر نشوند. به‌علاوه، به این نکته واقف خواهیم بود که این کار هزینه‌ای در زمان اجرا به ما تحمیل خواهد کرد، به‌گونه‌ای که در نام‌گذاری فایل‌ها و کلاس‌ها می‌باید قوانینی را رعایت کنیم و بسیاری موارد دیگر. پس همان‌طور که ملاحظه شد، یک نام ساده می‌تواند اطلاعات تفصیلی زیادی، حتی تا سطح پیاده‌سازی در اختیار ما قرار دهد. اما زمانی که از الگوی معماری صحبت می‌کنیم، با نام آن، اطلاعات مفیدی در مورد مشخصه‌های سیستم به دست خواهیم آورد، اما این بار اطلاعات چندان جزئی نخواهند بود به‌عنوان مثال زمانی که می‌گوییم الگوی معماری یک نرم‌افزار لوله و فیلتر است، همان‌طور که در ادامه اشاره خواهد شد، در مورد اینکه اجزای نرم‌افزار چه ویژگی‌هایی خواهند داشت، نحوه ارتباط میان این اجزا چگونه خواهد بود، انتقال کنترل اجرا در سطح نرم‌افزار به چه شکل صورت خواهد پذیرفت و مواردی از این قبیل، دید واضحی به دست می‌آوریم؛ اما هیچ‌کدام از این موارد به‌دقت و ریزدانگی مواردی که در مورد الگوی طراحی ذکر شد، نیستند (Yu et al., 2017).

۳-۳- الگوی میکروسرویس[2]

معماری میکروسرویس‌ها مبتنی بر یک فلسفه اشتراک - هیچ[3] است و به یک تجربه طولانی در مهندسی نرم‌افزار و طراحی سیستم متکی است. این سبک معماری یک سیستم را به‌صورت مجموعه‌ای از سرویس‌های کوچک به‌هم پیوسته می‌سازد که در واحدهای کوچک منسجم و مستقل جدا شده‌اند (Ghofrani & Lübke, 2018).

گسترده‌ترین تعریف از معماری میکروسرویس‌ها «رویکردی برای توسعه یک برنامه واحد به‌عنوان مجموعه‌ای از سرویس‌های کوچک است که هر کدام در فرایند خود اجرا می‌شوند و با مکانیزم‌های سبک و اغلب با یک HTTP API ارتباط برقرار می‌کنند.» برخلاف مونولیت‌ها، میکروسرویس‌ها قابلیت گسترش و مقیاس‌پذیری مستقل را پرورش می‌دهند و می‌توانند با استفاده از انواع فناوری‌های مختلف توسعه یابند (Garriga, 2018).

مارتین فاولر و جیمز لوئیز در مقاله‌ای در سال ۲۰۱۴ برای مجله رادار فناوری[4]، از اصطلاح میکروسرویس استفاده کردند. آنها گفتند که چون میکروسرویس‌ها توسعه‌دهندگان را قادر می‌سازند نرم‌افزار را به شیوه‌ای چابک‌تر و مقیاس‌پذیرتر از سیستم‌های یکپارچه ایجاد و استقرار دهند، راه‌حل برتری هستند.

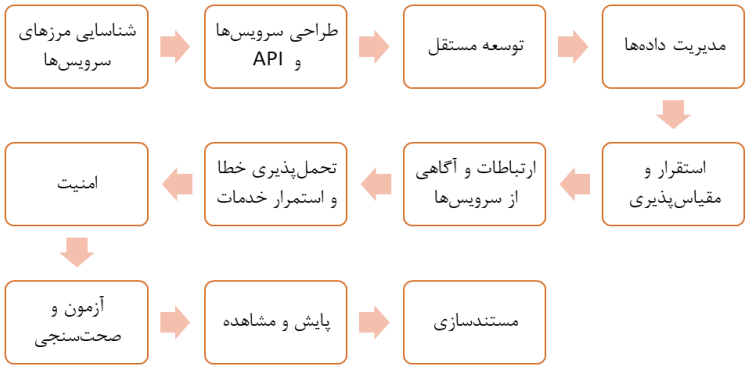
چابکی و استقلال میکروسرویس‌ها به منظور بهره‌مندی از مزایای این الگو، همواره مورد تاکید قرار گرفته است (Osman et al., 2019).

---

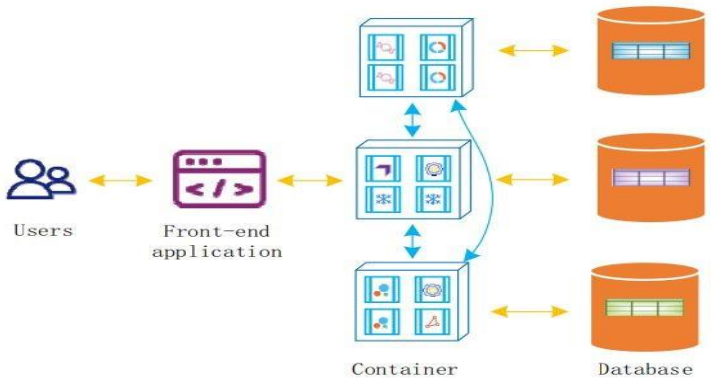
[1] Bridge
[2] Micro Service
[3] Share – Nothing
[4] www.thoughtworks.com

معماری میکروسرویس (MSA[1]) از معماری توسعه چابک سرچشمه می‌گیرد و در سال‌های اخیر پس از رشد معماری سرویس‌گرا در صنعت و دانشگاه مورد توجه قرار گرفته است. هیچ توافقی در مورد رابطه بین MSA و SOA حاصل نشده است. برخی از طرفداران MSA ادعا می‌کنند که MSA یک سبک معماری جدید است، درحالی‌که برخی از طرفداران SOA صرفاً یک رویکرد فکر می‌کنند که MSA پیاده‌سازی SOA است (Li et al., 2021).

در شکل ۵، برنامه میکروسرویس تقسیم سرویس برای رسیدن کسب‌وکار معین و قسمت‌های مختلف تحت سرور به چند می‌شود که هر به یک مسئولیت مدیریت اجرا در برنامه است.



شکل ۵ - نحوه پیاده‌سازی الگوی معماری میکروسرویس



شکل ۶ – نمایی از معماری میکروسرویس‌ها

پیاده‌سازی الگوی معماری میکروسرویس

پیاده‌سازی معماری میکروسرویس شامل طراحی و ساختن یک سیستم نرم‌افزاری به عنوان مجموعه‌ای از خدمات کوچک و مستقل است. هر سرویس بر روی یک قابلیت تجاری خاص تمرکز دارد و می‌تواند به طور مستقل توسعه یافته، مستقر شده و با استفاده از قابلیت‌های مقیاس‌پذیری، خدمات ارائه کند. در شکل ۵ - نحوه پیاده‌سازی الگوی معماری میکروسرویس، یک نمای کلی از نحوه پیاده‌سازی معماری میکروسرویس ها آورده شده است:

[1] Micro Service Architecture

مشخصه‌های الگوی معماری میکروسرویس

- **سرویس‌ها**: میکروسرویس‌ها، مولفه‌های فشرده، مستقل و ماژولار هستند که قابلیت‌های تجاری ارائه می‌دهند.
- **مدیریت غیرمتمرکز**: هر میکروسرویس یک تیم پیاده‌سازی با فرایندهای مجزا را می‌تواند داشته باشد.
- **مبتنی بر API**[1]: پروتکل‌های سبک همچون REST و پیام‌رسانی، برای ارتباط بین میکروسرویس‌ها استفاده می‌شود.
- **توسعه مستقل:** هر میکروسرویس مستقل از سایر میکروسرویس‌ها امکان توسعه و انجام تغییرات را دارد که منجر به افزایش سرعت انتشار نسخه‌ها و تسهیل نگهداری می‌شود.

چالش‌های معماری میکروسرویس (Jaramillo et al., 2016)

- **پیچیدگی:** توسعه، پایش و آزمون تعداد زیاد میکروسرویس‌ها زمان‌بر بوده و می‌تواند چالش‌زا باشد.
- **سیستم‌های توزیع‌شده:** ازآنجاکه سیستم‌های توزیع‌شده بخشی از معماری میکروسرویس‌ها هستند، توسعه‌دهندگان باید برای قطع شبکه برنامه‌ریزی کرده و مطمئن شوند که سرویس‌های می‌توانند ارتباطات ناهمزمان را تحمل کنند.
- **یکپارچگی**: ادغام میکروسرویس‌های چندین تیم مختلف، ممکن است امری دشوار باشد؛ چراکه مستلزم اطمینان از ارتباط مناسب و سازگاری آن‌ها است.
- **آزمون:** فرایند آزمون برای میکروسرویس‌ها، نیازمند زمان و تلاش زیادی است. هر میکروسرویس باید علاوه بر اینکه به طور مستقل آزمون می‌شود، باید هنگام ارتباط با سایر سرویس‌ها نیز مورد آزمون قرار بگیرد.

۳-٤- الگوی کلاینت – سرور[2]

معماری کلاینت – سرور یک الگوی معماری است که وظیفه‌ها و نقش‌های سیستم بین دو مؤلفه اساسی تقسیم می‌شود: کلاینت و سرور. در این مدل، کلاینت نقشی را بر عهده دارد که درخواست‌ها و عملیات‌های کاربر را انجام می‌دهد. این مؤلفه معمولاً مرتبط با رابط کاربری و تعامل کاربر با سیستم است. کلاینت می‌تواند یک برنامه کاربردی، مرورگر وب یا دستگاه متصل به شبکه باشد. از سوی دیگر، سرور مسئولیت پردازش و ذخیره‌سازی داده‌ها و ارائه خدمات را بر عهده دارد. سرور معمولاً به‌عنوان یک مرجع مرکزی عمل می‌کند و درخواست‌های کلاینت را پردازش می‌کند و نتایج را به کلاینت ارسال می‌کند. سرور می‌تواند یک سرور فیزیکی یا یک خدمت ابری باشد. معماری کلاینت – سرور به چهار نوع طبقه‌بندی می‌شود (Kumar, 2019):

– معماری یک لایه، معماری دو لایه، معماری سه لایه و معماری N لایه.

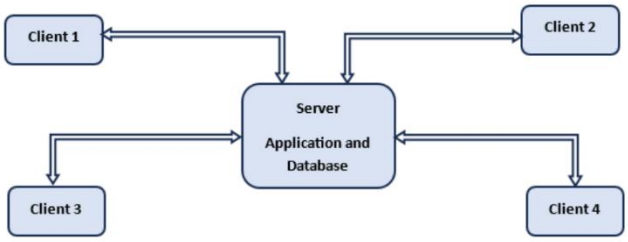
این الگوی معماری، بر اساس مفهوم محاسبات توزیع‌شده ساخته شده است که در آن وظایف مختلف به دستگاه‌ها یا سیستم‌های مختلف اختصاص داده می‌شود که منجر به پردازش سریع‌تر و افزایش کارایی می‌شود. یک پروتکل شبکه این ارتباط را امکان‌پذیر می‌کند (Nyabuto, 2024).

---

[1] Application Program Interface
[2] Client – Server

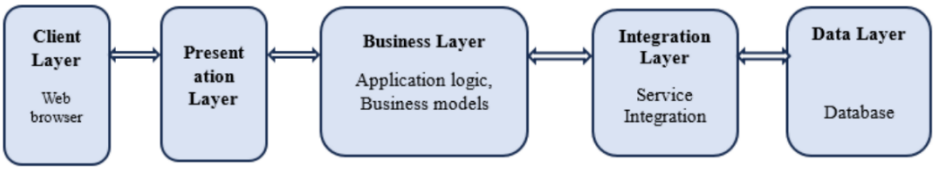
در شکل ۵- معماری دولایه‌ی کلاینت – سرور، شکل ۷ - معماری N لایه‌ی کلاینت – سرور و در شکل ۷ - معماری N لایه‌ی کلاینت – سرور (Nyabuto, 2024) نشان داده شده و ویژگی‌های اصلی آن‌ها در جدول ۱، بررسی شده است.

در معماری کلاینت سرور، کلاینت‌ها را می‌توان به دو دسته تقسیم‌بندی نمود: معماری با کلاینت لاغر[1] یا ضعیف و معماری با کلاینت ضخیم[2]. در دسته‌ی اول، کلاینت تقریبا هیچ کار پردازشی انجام نمی‌دهد و تماما به سرور و منابع آن وابسته است (Ahlan et al., 2010)؛ از این روش در مواردی که داده‌ها حساس بوده و یا رایانه‌های کلاینت توان اجرا یا ذخیره‌سازی ندارند استفاده می‌شود (Abdullah et al., 2017). در دسته‌ی دوم، کلاینت‌ها، امکان پردازش دارند و می‌توانند برخی از درخواست‌ها را بدون استفاده از سرور و یا با همکاری آن، پردازش نموده، برنامه‌هایی اجرا کنند و به درخواست‌ها پاسخ دهند (Zakoldaev et al., 2019).



شکل ۵- معماری دولایه‌ی کلاینت – سرور (Nyabuto, 2024)

شکل ۶- معماری سه‌لایه‌ی کلاینت – سرور (Nyabuto, 2024)



شکل ۷ - معماری N لایه‌ی کلاینت – سرور (Nyabuto, 2024)

[1] Thin
[2] Thick

جدول ۱ - مقایسه انواع معماری کلاینت - سرور

الگوی معماری کلاینت - سرور	مزایا	معایب
دولایه	• کارایی بالا، به دلیل یکجا بودن پایگاه داده و برنامه اصلی • سهولت توسعه سیستم با توجه به عدم نیاز به ارتباط با سایر سرورها • همگن بودن برنامه‌های ساخته شده و دارای منطق ایستا	• کارایی در هنگام افزایش کاربران و استفاده از منابع زیاد، چالش‌زا خواهد بود. • قابلیت حمل پایین سیستم با توجه به اتصالات محکم دو بخش
سه‌لایه	• قابلیت نگهداری بالا • مقیاس‌پذیری بهبود یافته • امنیت بالاتر	• پیچیدگی توسعه در مقایسه با دولایه، به افزایش نقاط ارتباطی • نیاز به توسعه سرور واسط و احتمال افزایش ترافیک
N لایه	• مقیاس‌پذیری بالا	• افزایش ارتباطات و ترافیک شبکه

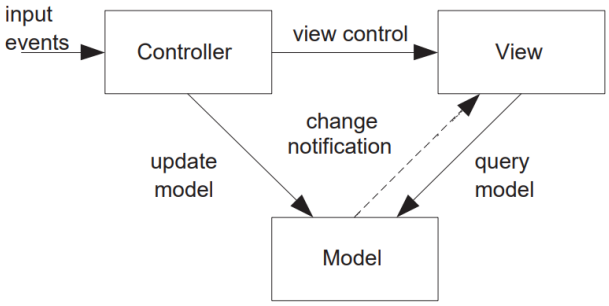
۳-۵- الگوی کنترل‌کننده- نمایش- مدل [1]

معماری MVC یک الگوی معماری محبوب در توسعه نرم‌افزار است که به طور گسترده در بسیاری از فریم‌ورک‌ها و پلتفرم‌ها استفاده می‌شود. این معماری، وظایف سیستم را بین سه مؤلفه اصلی تقسیم می‌کند: مدل[2]، نمایش[3] و کنترل‌کننده[4] (Jailia et al., 2016).

- مدل: مسئولیت برقراری ارتباط با داده‌ها و منطق کسب‌وکار را بر عهده دارد. این مؤلفه شامل استخراج، ذخیره، به‌روزرسانی و حذف داده‌ها است. مدل به‌صورت مستقل از سایر مؤلفه‌ها وابستگی دارد و وظیفه‌ای ندارد که به طور مستقیم با کاربر تعامل داشته باشد. معمولاً استفاده از الگوهای طراحی دیگری مانند مخازن[5] و سرویس‌ها در این مؤلفه معمول است.
- نمایش: مسئولیت نمایش و ارائه داده‌ها به کاربر را دارد. این مؤلفه وظیفه نمایش داده‌ها و دریافت ورودی کاربر را بر عهده دارد. نمایش می‌تواند شامل واسط کاربری، صفحات وب، رابط‌های گرافیکی و غیره باشد. نمایش باید به‌صورت مستقل از مدل و کنترل‌کننده عمل کند.

[1] Model-View-Controller (MVC)
[2] Model
[3] View
[4] Controller
[5] Repositories

- کنترل‌کننده: مسئولیت کنترل جریان برنامه و ارتباط میان مدل و نمایش را بر عهده دارد. این مؤلفه پاسخگوی درخواست‌ها و رویدادهای کاربر است و بر اساس آن‌ها عملیات‌های لازم را بر روی مدل انجام می‌دهد و نتیجه را به نمایش می‌دهد. نمایی از الگوی معماری MVC در شکل ۶ نشان داده شده است.



شکل ۸ – نمایی از الگوی معماری MVC

نحوه کار دو مولفه‌ی کنترل‌کننده و نمایش بسیار وابسته به این است که رابط کاربری مبتنی بر متن (TUI[1]) طراحی شده است یا مبتنی بر رابط کاربری گرافیکی. در رابط کاربری مبتنی بر متن نیاز است، کاربر به‌وسیله‌ی گام‌های موردنیاز برای تکمیل فرایند پردازش راهنمایی شود. از منظر ورود داده‌ها، یک TUI، کاربر را مجبور می‌کند تا داده‌ها را به ترتیب از پیش تعیین‌شده وارد کند. یک رابط کاربری گرافیکی به نرم‌افزار اجازه می‌دهد تا اقداماتی را ارائه دهد که کاربر می‌تواند از بین آنها انتخاب کند و اجازه می‌دهد تا وارد کردن داده‌ها توسط کاربر (معمولا) به هر ترتیبی انجام شود (Voorhees, 2020).

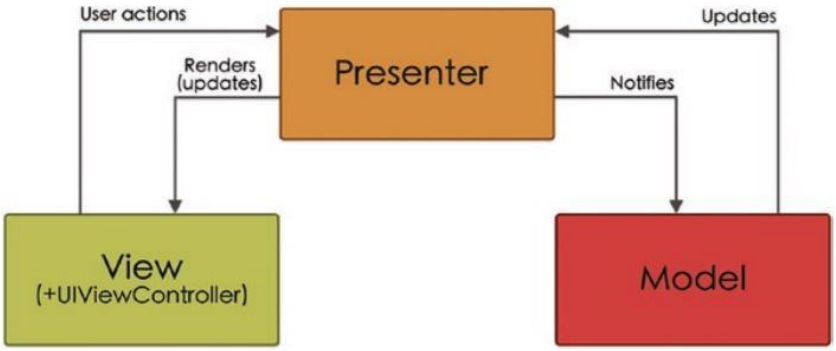
به طور معمول هر کنترل‌کننده، یک نمایش را هدایت می‌کند، در حالت هدایت چند نمایش، توسعه‌دهنده مسئول مدیریت این موضوع است. این الگو، برای کنترل حالت نمایش‌ها ضعیف عمل می‌کند این مشکل در الگوی MVVM[2] برطرف شده است. تغییر در مدل باعث تغییر کنترل‌کننده و نمایش می‌شود که با توجه به هزینه بروزرسانی مکرر باعث، ممکن است منجر به افت سرعت شود (لطفی و زمانی, ۱۳۹۴).

۳-۶- الگوی ارائه‌دهنده- نمایش - مدل[3]

MVP یک الگوی طراحی وب است که از اجزای Model، View و presenter استفاده می‌کند. همانند MVC و MVVM، این مدل نشان‌دهنده شیء دامنه و View رابطی است که باید به کاربر نمایش داده شود. بااین‌حال مؤلفه ارائه‌دهنده برای پردازش ورودی کاربر، همگام‌سازی مدل و نما، و اصلاح نما در هر زمان که لازم باشد استفاده می‌شود. مزیت پیاده‌سازی MVP این است که مجری کنترل کاملی بر مدل و نما دارد؛ بنابراین، presenter قادر است View را بر اساس حالت‌ها و قابلیت‌های مختلف تغییر دهد. به‌عنوان‌مثال، همان‌طور که ارائه‌دهنده، خطا در مدل را تشخیص می‌دهد، ارائه‌دهنده می‌تواند نما را برای رسیدگی به وضعیت خطا در مدل کنترل کند. نمایی از الگوی معماری MVP در شکل ۷ نشان داده شده است (Anuar et al., 2017).
توضیحات سه لایه مدل، نمایش و ارائه‌دهنده به‌صورت زیر می‌باشد (García, 2023):

[1] Text-Based User Interface
[2] Model-View-Viewmodel (MVVM)
[3] Model-View-Presenter (MVP)

- مدل: در الگوی MVP، مدل (مؤلفه) مسئول منطق کسب‌وکار و ذخیره‌سازی، دستکاری و دسترسی به داده‌های برنامه است.
- در MVP، لایه مدل تنها می‌تواند با لایه ارائه‌دهنده ارتباط برقرار کند (مدل از وجود یک نما بی‌اطلاع است)
- نمایش: در مدل MVP، لایه View بر خلاف الگوی MVC، لایه نمایش شامل هر دو مؤلفه View (UIView) و Controller (UIViewController) است. همچنین، در الگوی MVP، هم View و هم Controller منطق بسیار کمتری نسبت به مدل MVC ذخیره می‌کنند که آن‌ها را سبک‌تر می‌کند.
- مجری یا ارائه‌دهنده: مجری مسئول دریافت رویدادهای رخ داده در نمایش و انتقال آن‌ها به مدل است. از سوی دیگر، ارائه‌دهنده نیز مسئول به‌روزرسانی نمایش در زمان تغییر داده‌ها است.

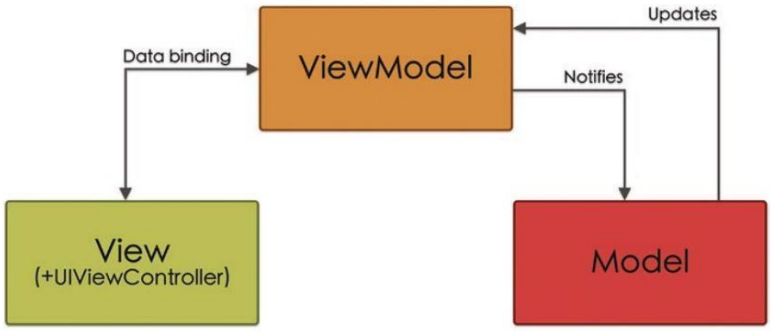


شکل ۹ – نمایی از الگوی معماری MVP

در واقع، تفاوت اصلی این است که منطق تجاری از کنترل‌کننده به ارائه‌دهنده منتقل شده است. این الگو نسبت به MVC در تفکیک بهتر مسئولیت‌ها و آزمون بهتر منطق کسب‌وکار، برتری دارد. لکن باید برای برخی از مشکلاتی همچون نیاز به کد بیشتر در توسعه، پیچیدگی بیشتر نسبت به MVC و احتمال بزرگ‌تر شدن ارائه دهنده، راهکار ارائه شود (Qureshi & Sabir, 2014).

۳-۷- الگوی MVVM

معماری MVVM تکاملی از معماری MVP است که در آن لایه ارائه‌دهنده با لایه ViewModel جایگزین می‌شود. علاوه بر این، برخی از مشکلات ارائه شده توسط معماری MVP را حل می‌کند. مهم‌ترین آن‌ها جفت‌شدگی است که در معماری MVP بین نما و ارائه‌دهنده وجود دارد. در مورد معماری MVVM، مدل نمایش (ViewModel) هیچ اشاره‌ای به نما (View) ندارد، یعنی هیچ جفت‌سازی وجود ندارد. نمایی از الگوی معماری MVVM در شکل ۸ نشان داده شده است (Sheikh & Sheikh, 2020).



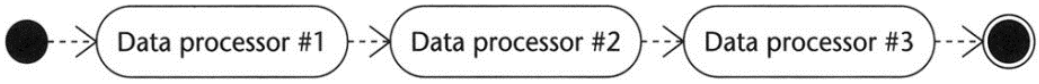
شکل ۱۰ – نمایی از الگوی معماری MVVM

۳-۸- الگوی لوله و فیلتر[1]

معماری لوله و فیلتر یک الگوی طراحی نرم‌افزاری است که تعدادی از اجزای پردازش‌کننده جریان داده را به‌هم متصل می‌کند. اتصال بین اجزاء در این سبک از معماری به‌وسیله‌ی یک لوله است. این معماری از فن اتصال خروجی برنامه به ورودی‌های دیگر در سیستم عامل یونیکس الهام گرفته شده است و معمولاً در سیستم‌هایی که نیاز به تبدیل یا تجزیه و تحلیل جریان‌های داده دارند، استفاده می‌شود.

اجزای اصلی این معماری، لوله‌ها و فیلترها هستند که لوله‌ها برقرارکننده ارتباط میان فیلترها هستند و داده‌ها و اطلاعات را جابه‌جا می‌کنند. فیلترها عناصری هستند که وظیفه پردازش داده‌های ورودی و تبدیل آنها به اطلاعات خروجی را بر عهده دارند. در اینجا فیلتر به نماد پردازشی اشاره دارد و از لوله‌ها برای اتصال فیلترها به‌گونه‌ای استفاده می‌شود که خروجی یک فیلتر به‌عنوان ورودی به فیلتر بعدی عمل کند. به هر مسیر جریان‌داده در یک نرم‌افزار مبتنی بر سبک لوله و فیلتر یک خط لوله گفته می‌شود. در بین دو فیلتر ممکن است برخی عناصر واسطه وجود داشته باشند که می‌توانند به‌عنوان بافر برای مدیریت جریان اطلاعات عمل کنند. مسیر جریان داده در این الگو، یک‌طرفه است که در شکل ۱۱ نشان داده شده است (Sharma et al., 2015).

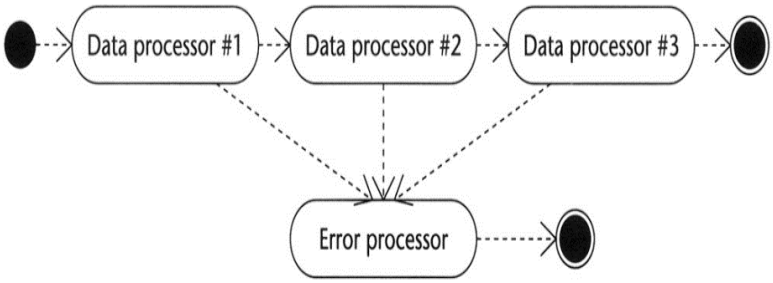
این الگو برای سرعت‌بخشیدن به پردازش استفاده می‌شود، زیرا چندین پردازش به‌صورت هم‌زمان در چندین فیلتر مختلف در حال انجام هستند.



شکل ۱۱- مسیر جریان داده

یک مدل ساده از این سبک، این است که تنها یک مسیر برای جریان داده یا تنها یک خط لوله وجود داشته باشد. شکل ۱۳، نمونه‌ای از خط لوله با فیلترهای دو درگاهی را نشان می‌دهد. البته نسخه دیگری از فیلترها که سه‌درگاه دارند، بسیار متداول‌تر است. در این مدل، از درگاه سوم معمولاً به عنوان خروجی برای گزارش خطا استفاده شده و این خروجی را به یک فیلتر مجزا که وظیفه آن رسیدگی به خطاها است، متصل می‌کند. شکل ۱۲- شمای کلی سبک لوله و فیلتر با فیلترهای سه درگاهی را نمایش می‌دهد.

[1] Pipe & Filter



شکل ۱۲- شمای کلی سبک لوله و فیلتر با فیلترهای سه درگاهی

از ویژگی‌های این سبک این است که هر فیلتر به محض دریافت ورودی، شروع به پردازش کرده و قبل از اینکه ورودی کاملاً مصرف شود، تولید خروجی آغاز می‌گردد. یعنی هر عنصر پردازشگر مانند فیلتری عمل می‌کند که در مسیر یک جریان داده سیال قرار گرفته و تبدیلاتی بر روی آن انجام می‌دهد. فیلترها معمولاً بر روی هم اثر نمی‌گذارند، مگر از طریق داده‌ای که یک فیلتر بالایی برای یک فیلتر پایینی ارسال می‌نماید. به همین جهت عمدتاً در منابع ذکرشده که فیلترها، حالات خود را به اشتراک نمی‌گذارند. علاوه‌براین، یک فیلتر ممکن است از وجود فیلترهای دیگر بی‌اطلاع باشد. به این ترتیب هر فیلتر تنها لازم است بداند که انتظار دیدن چه نوع ورودی را دارد. به لوله‌های استفاده‌شده در این سبک، می توان نوع اختصاص داد. متداول‌ترین انواع لوله به شرح زیر است:

لوله‌های جریانی[1]: گونه‌ای از لوله‌ها که هر جریان از بیت‌های دودویی را انتقال می‌دهد. فیلترهایی که از این لوله‌ها داده دریافت می‌کنند، لازم است ابتدا بر روی داده‌های دریافتی پیش‌پردازش انجام دهند (داده‌ها سریال هستند و باید تبدیل به اشیاء قابل تشخیص نظیر اعداد صحیح، رشته‌های حرفی، آرایه‌ها و غیره شوند). همچنین، فیلترهایی که خروجی خود را روی این نوع لوله‌ها قرار می‌دهند، می‌بایست ابتدا این خروجی را به قالب سریال درآورند (bass et al, 2003).

لوله‌های شئ[2]: لوله‌ای را گویند که در آن اشیاء منتقل می‌شوند. در اینجا منظور از شئ، کلی‌تر از آن است که در بحث شئ‌گرایی می‌شناسیم و به معنای دسته‌بندی داده‌ها است که دربرگیرنده یک مفهوم مشخص (و انتزاعی) است.

در لوله‌های نوع دوم، یک شئ به‌صورت کلی در فیلتر مصرف شده و یا روی یک خط لوله جابه‌جا می‌شود و امکان شکستن آن به اشیا کوچک‌تر وجود ندارد. در استفاده از انواع لوله‌ها باید دقت داشت که به‌کارگیری بیش از حد یا نابه‌جای هر کدام، می‌تواند به کارایی کلی سیستم لطمه وارد کند. به‌عنوان نمونه استفاده محض از لوله‌های شئ، سبب می‌شود سیستم به یک سیستم پردازش دسته‌ای تبدیل شود و دیگر جریان داده‌ها که زمینه‌ساز اصلی ویژگی‌های خوب این سبک نرم‌افزارها است، از بین برود. بعضی اوقات و به دلیل ملاحظات و نیازمندی‌های خاص نرم‌افزار بر روی حجمی از اطلاعات که یک لوله می‌تواند حمل کند،

[1] Stream-typed Pipe
[2] Object-typed Pipe

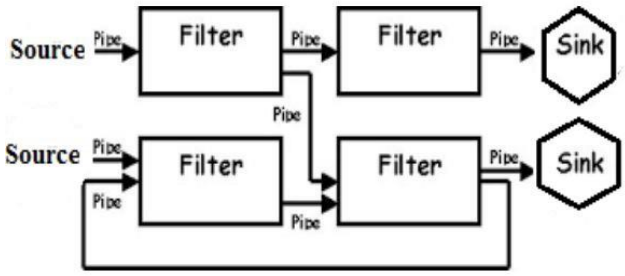
محدودیت اعمال می‌شود. این کار به‌عنوان مثال، می‌تواند به دلیل محدودیت پهنای باند در یک سیستمی صورت پذیرد که بخشی از پردازش‌ها روی شبکه انجام می‌شود، و فیلترها روی پردازنده‌های مجزا توزیع شده‌اند.

مزایای الگوی لوله و فیلتر

در این الگو رفتار کل سیستم به صورت ترکیب ساده‌ای از رفتار فیلترها مدل می‌شود. به دلیل استقلال فیلترها از هم، توسعه سیستم با سهولت انجام می‌شود. استفاده مجدد از نرم افزار توسط این سبک پشتیبانی می‌شود. الگوی لوله‌ها و فیلترها به طور طبیعی از اجرای موازی کارها پشتیبانی می‌کنند. برای موازی‌سازی یک عمل، باید ابتدا قسمت‌هایی از آن که امکان موازی شدن دارند را یافت، سپس به ازای هر کدام از این بخش‌ها یک خط لوله به سیستم اضافه کرد.

معایب الگوی لوله و فیلتر

به طور ذاتی این سبک تمایل دارد به سمت سیستم‌های پردازش دسته‌ای سوق پیدا کند و همان‌طور که قبلاً گفته شد، با از میان رفتن مفهوم جریان داده در نرم‌افزار (امکان تولید خروجی قبل از مصرف کامل ورودی) افت کارایی قابل پیش‌بینی است. در صورت استفاده بیش از حد از لوله‌های جریانی هم افت کارایی خواهیم داشت. هر بار که یک جریان داده دودویی به یک فیلتر می‌رسد، آن فیلتر می‌بایست ابتدا داده‌ها را پیش‌پردازش کند تا بتواند بر روی آنها پردازش اصلی را صورت دهد و این کار سبب افت کارایی خواهد شد. به دلیل آنکه نرم‌افزارهایی که با این سبک ساخته می‌شوند، ذاتاً به تبدیلات متنوع و مکرر داده‌ها بستگی دارند، معمولاً این سبک برای نرم‌افزارهای محاوره‌ای انتخاب مناسبی نیست (Bird et al, 2015).



شکل ۱۳ - نمایی از الگوی معماری لوله و فیلتر

۳-۹- الگوی طراحی دامنه- محور[1]

الگوی طراحی دامنه محور، مبحثی است که در سال‌های اخیر مورد توجه بسیاری از شرکت‌های توسعه‌دهنده‌ی نرم‌افزار گرفته و رویکرد آن‌ها را برای تحلیل، طراحی، ساخت و نگهداری نرم‌افزارها به خصوص نرم‌افزارهای بزرگ و پیچیده، به شدت تحت تاثیر قرار داده است (Tune & Millett, 2015).

در پروژه‌های نرم‌افزاری بزرگ، معمولاً گروه‌های ذینفع به‌عنوان مثال واحدهای تجاری مختلف، درگیر هستند. این گروه‌ها به زبان‌های مختلف تجاری صحبت می‌کنند و الزامات مختلفی را برای مدل دامنه تدوین می‌کنند. این الگو رویکردی است برای طراحی و توسعه نرم‌افزارهایی که دارای فرآیندهای پیچیده و قوانین زیاد بوده و مکرراً در حال تغییر هستند. اصطلاح دامنه به

[1] Domain Driven Design (DDD)

حوزه فعالیتی گفته می‌شود که نرم‌افزار برای پیاده‌سازی آن توسعه می‌یابد. بنیان این الگو مجموعه‌ای از مفاهیم و فنونی است که برای تجزیه و تحلیل دامنه و ساخت یک مدل متناظر از روی آن به کار برده می‌شود. الگوی طراحی دامنه -محور، همراه با جزئیات دقیق بوده و تمام مفاهیم و قوانین کسب‌وکار موردنیاز در آن پیاده‌سازی می‌شود. از این الگو بیشتر برای سیستم‌های سازمانی که کسب‌وکار آنها پیچیده و همچنین طول عمر آنها بالا است، استفاده می‌شود. تلاش برای حفظ یک مدل جهانی اغلب در عمل با شکست مواجه می‌شود؛ زیرا این مدل‌ها بیش از حد پیچیده و مبهم می‌شوند. ابهام، اغلب زمانی به وجود می‌آید که گروه‌های ذینفع مختلف اصطلاحات یکسانی را در زبان تجاری خود دارند، اما آن را با معانی متفاوتی به کار می‌برند؛ بنابراین، یک مفهوم مرکزی الگوی طراحی دامنه -محور، «زمینه‌های محدود[1]» است (Braun et al., 2021).

یک زمینه محدود، یک مرز واضح را تعریف می‌کند که در آن یک مدل دامنه یکپارچه خاص معتبر است. همچنین مشخص می‌کند که کدام زبان تجاری در این زمینه معتبر است. مهم است که این زبان کاملاً تعریف شده توسط متخصصان دامنه و همچنین مهندسان نرم‌افزار صحبت شود و همچنین این زبان به طور مداوم در مدل و کد استفاده شود. ایجاد زبان یکسان بین دو تیم، در مورد مفاهیم دامنه امری الزامی است؛ بنابراین، الگوی طراحی دامنه -محور از آن به‌عنوان یک زبان فراگیر یاد می‌کند. یک زمینه محدود، دارای پایگاه کد و پایگاه داده خاص خود است که توسط یک تیم نگهداری می‌شود و همچنین راه‌کارهایی برای تقسیم نرم‌افزار به بخش‌های جدا، مستقل و همچنین ارتباط این بخش‌ها با یکدیگر ارائه می‌کند. فرآیند توسعه‌ی نرم‌افزار به‌صورت موازی بین چند تیم انجام شده و معماران سیستم را قادر می‌سازد تا از معماری‌ها و فناوری‌های مختلف در بخش‌های مختلف استفاده نمایند. بیشتر راه‌حل‌هایی که در این الگو مطرح می‌شود، برای ساده‌سازی و جداسازی یک دامنه پیچیده به بخش‌های کوچک‌تر و ارائه‌ی راه‌حل برای هرکدام از آن قسمت‌ها است. در معماری میکروسرویس، یک زمینه محدود معمولاً با یک میکروسرویس مطابقت دارد. در بخش طراحی راهبردی این الگو، با برش زمینه محدود مناسب مواجه می‌شویم. بخش طراحی تاکتیکی به طراحی مدل دامنه می‌پردازد. مهم‌ترین مفاهیم بخش طراحی تاکتیکی عبارت‌اند از (Evans, 2004) :

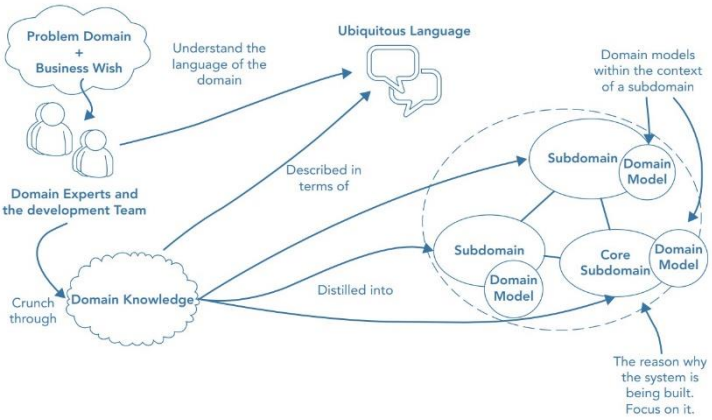
- موجودیت‌ها[2]: اشیاء دامنه با هویت و چرخه حیات کاملاً تعریف شده (به‌عنوان مثال، یک شیء مشتری)
- اشیاء ارزش[3]: اشیاء دامنه‌ای که فقط بر اساس مقادیر ویژگی‌هایشان قابل تشخیص هستند (به‌عنوان مثال، یک شی قیمت)
- تجمیع[4]: خوشه‌هایی از اشیاء دامنه منسجم که باید به طور هم‌زمان به‌روز شوند. تجمیع دارای یک موجودیت ریشه اختصاصی است که مسئول حفظ ثبات قوی و متغیرهای دامنه درون توده است.
- عملیات دامنه[5]: روش‌هایی در اشیاء دامنه که منطق تجاری را محصور می‌کنند (منطق دامنه به زبان طراحی دامنه-محور)
- خدمات[6]: در مواردی که منطق دامنه مسئولیت طبیعی یک شیء دامنه نیست، خدمات می‌توانند معرفی شوند.
- مخازن[7]: رابط‌هایی برای پرس‌وجو، درج و حذف مجموعه‌ها از پایگاه‌داده.

---

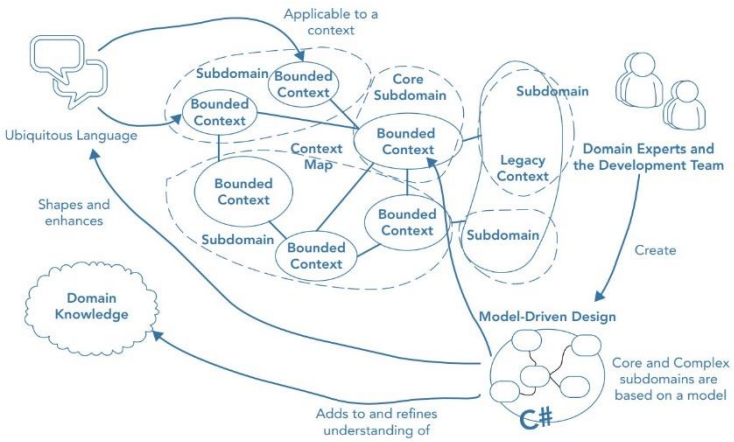
[1] Bounded Contexts
[2] Entities
[3] Value Objects
[4] Aggregates
[5] Domain Operations
[6] Services
[7] Repositories

- رویدادهای دامنه[1]: اشیاء دامنه نشان‌دهنده رویدادهای قابل توجه در دامنه هستند که کارشناسان یا کاربران دامنه می‌خواهند در مورد آنها مطلع شوند.

در الگوی طراحی دامنه-محور، الگوها به مدیریت پیچیدگی یک مسئله کمک می‌کنند - به نام فضای مشکل یا پیچیدگی را در راه‌حل مدیریت می‌کنند - به نام فضای راه‌حل. تاکید این الگو بر شناخت خوب مسئله و بعد ارائه راه‌حل است. فضای مشکل، همان‌طور که در شکل ۱۶ نشان داده شده است، دامنه مشکل را به زیردامنه‌های قابل مدیریت‌تر تقسیم می‌کند. زیردامنه‌های موجود در فضای مساله در فضای راه‌حل به BC [2]تبدیل می‌شوند. در حالت ایده‌آل هر زیردامنه در پیاده‌سازی به یک BC تبدیل می‌شود. هر BC یک واحد مستقل شناخته می‌شود و مدل دامنه مخصوص به خود را دارد. الگوهای طراحی و معماری در هر BC می‌تواند بسته به ماهیت آن متفاوت باشد. تأثیر الگوی طراحی دامنه-محور در فضای مشکل این است که نشان دهد چه چیزی مهم است و کجا باید تلاش را متمرکز کرد. سمت راه‌حل این الگو که در شکل ۱۷ نشان داده شده است، الگوهایی را پوشش می‌دهد که می‌توانند معماری برنامه‌ها را شکل دهند و مدیریت آن را آسان‌تر کنند (Tune & Millett, 2015).



شکل ۱۶ - الگوهای طراحی دامنه-محور قابل اجرا برای فضای مسئله



شکل ۱۷ - الگوهای طراحی دامنه-محور قابل اجرا برای فضای راه‌حل

[1] Domain Events
2 Bounded Context

۳-۱۰- الگوی رویداد-محور[1]

سیستم‌های نرم‌افزاری سازمانی داده‌ها را از یک سیستم به سیستم دیگر از طریق شبکه‌ای مانند شبکه محلی[2] یا شبکه گسترده[3] ارسال می‌کنند. این داده‌ها طبق یک پروتکل یا استاندارد خاص به‌عنوان پیام دسته‌بندی می‌شوند و بین سیستم‌ها و برنامه‌های کاربردی در سیستم نرم‌افزار سازمانی به اشتراک گذاشته می‌شوند. این پیام ممکن است حاوی داده‌هایی باشد که در صورت معامله فروش می‌تواند میلیون‌ها دلار ارزش داشته باشد. از سوی دیگر، می‌تواند حاوی برخی از داده‌های نامعتبر باشد که توسط یک کاربر جعلی ارسال شده است. درک ارزش داده‌ها و طراحی سیستم به‌گونه‌ای مهم است که انتظارات کسب‌وکار را برآورده کند. در حالت اول، ازدست‌دادن آن پیام (داده) به معنای زیان بزرگی برای شرکت است.

اما در مورد دوم، اصلاً مسئله‌ای نیست. این بدان معناست که هنگام طراحی یک سیستم نرم‌افزاری سازمانی باید ضمانت داده را شناسایی کنیم. ارتباطات هم‌زمان می‌تواند با اجرای دوباره تلاش‌ها در سمت مشتری، الزامات تضمین داده را برطرف کند. اما این کار پیاده‌سازی مشتریان را پیچیده می‌کند. در عوض، مدل «ارتباط غیرهم‌زمان[4]» می‌تواند این مشکل را به روشی بسیار ساده‌تر با معرفی یک واسطه پیام[5] و جداکردن مشتری[6] (فرستنده) و سرور[7] (گیرنده) حل کند. در این مدل از پیام‌ها به‌عنوان رویداد یاد می‌شود و به همین دلیل معماری رویدادمحور نامیده می‌شود (Fernando, 2022).

معماری رویدادمحور سیستم را از نظر تولید، پردازش، نظارت و خاتمه رویدادهایی که می‌تواند در یک سیستم رخ دهد، نشان می‌دهد. در اینجا اصطلاح رویداد به «تغییر وضعیت» اشاره دارد. معماری رویدادمحور همان‌طور که در شکل ۱۶ نشان داده شده است، شامل چهار لایه منطقی یا عنصر معماری است (Sharma et al., 2015):

۱. تولیدکننده رویداد[8]: این لایه وظیفه تولید رویداد را بر عهده دارد. یعنی منبع رویداد در این لایه قرار دارد. منبع رویداد می‌تواند کلیک ماوس، فشاردادن کلید صفحه‌کلید، سرویس‌گیرنده ایمیل و غیره باشد.

۲. کانال رویداد[9]: برای انتقال رویداد از منبع آن به مصرف‌کننده رویداد موردنیاز است؛ یعنی موتور رویداد یا سینک. پس از اینکه رویدادهای نسل در یک صف ذخیره می‌شوند، سپس با کمک کانال رویداد به سینک منتقل می‌شوند. کانال رویداد می‌تواند یک اتصال TCP/IP یا هر فایل ورودی مانند فرم XML و غیره باشد.

۳. موتور پردازش رویداد[10]: مکانی است که رویدادها پردازش می‌شوند و پاسخ‌های مناسب باتوجه‌به رویداد تولید می‌شوند.

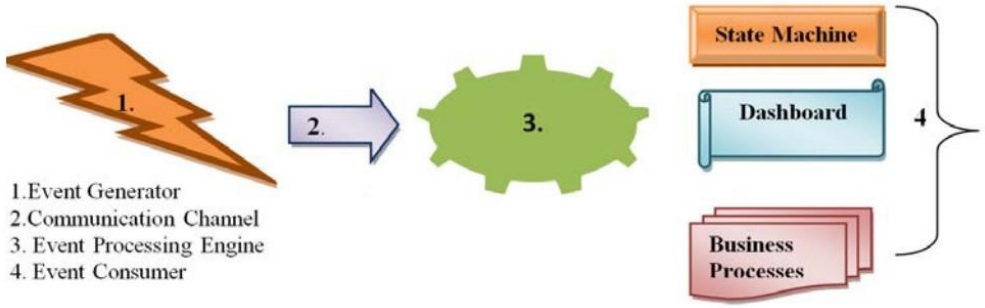
۴. فعالیت رویدادمحور پایین‌دست[11]: در اینجا پیامدهای رویدادها نشان داده شده است. به‌عنوان‌مثال، ارسال نامه بدون موضوع، پیام هشدار را نشان می‌دهد؛ بنابراین ارسال ایمیل یک رویداد است و پیام خطا، پیامد رویداد را نمایش می‌دهد.

EDA[12] یک الگوی طراحی انعطاف‌پذیر و مقیاس‌پذیر است که به‌ویژه برای توسعه سیستم‌های توزیع شده مناسب است (Gilbert & Price, 2021; Kleppmann et al., 2019). در میان ویژگی‌های دیگر، مزیت EDA در جداسازی اجزا و ماژولار بودن و قابلیت

---

[1] Event Driven
[2] Local Area Network (LAN)
[3] Wide Area Network (WAN)
[4] Asynchronous communication
[5] Message broker
[6] Client
[7] Server
[8] Event generator
[9] Event Channel
[10] Event processing engine
[11] Downstream eventdriven activity
[12] Event-Driven Architecture

استفاده مجدد است. تولیدکنندگان و مصرف‌کنندگان نیازی به دانستن وجود یکدیگر ندارند، که این امر باعث می‌شود که اجزای جدید را بدون تأثیر بر اجزای موجود آسان‌تر به سیستم اضافه کنند. همراه با منبع‌یابی رویداد، EDA تکرارپذیری و شفافیت را نیز امکان پذیر می‌کند (Krause et al., 2023).



شکل ۱۸ – معماری مبتنی بر رویداد

۳-۱۱- الگوی GOF[1]

Gangs of Four Design Patterns مجموعه‌ای از ۲۳ الگوی طراحی از کتاب "Reusable Object-Oriented Software" است. این کتاب اولین‌بار در سال ۱۹۹۴ منتشر شد و یکی از محبوب‌ترین کتاب‌ها برای یادگیری الگوهای طراحی است. این کتاب توسط اریش گاما، ریچارد هلم، رالف جانسون و جان ولیسیدز نوشته شده است. به دلیل چهار نویسنده به الگوهای طراحی گروه چهار گرفت. علاوه بر این، نام کوتاه‌تری به‌عنوان «الگوهای طراحی GoF» داشت. الگوهای طراحی GoF به سه دسته تقسیم می‌شوند (Gamma et al., 2015):

۱. خلاقانه[2]: الگوهای طراحی که با ایجاد یک شیء سروکار دارند. ۵ الگوی طراحی خلاقانه در جدول ۱ نشان داده شده است.

۲. ساختاری[3]: الگوهای طراحی در این دسته با ساختار طبقاتی مانند وراثت و ترکیب سروکار دارند. ۷ الگوی طراحی ساختاری در جدول ۲ نشان داده شده است.

۳. رفتاری[4]: این نوع الگوهای طراحی راه‌حلی را برای تعامل بهتر بین اشیاء، نحوه ارائه اتصالات ازدست‌رفته و انعطاف‌پذیری برای گسترش آسان در آینده ارائه می‌دهند. ۱۱ الگوی طراحی رفتاری در جدول ۳ نشان داده شده است.

جدول ۲ – الگوهای طراحی خلاقانه

ردیف	نام الگو	توصیف
۱	سینگلتون (Singleton)	مقداردهی اولیه یک کلاس را محدود می‌کند تا اطمینان حاصل شود که فقط یک نمونه از کلاس می‌تواند ایجاد شود.
۲	کارخانه (Factory)	مسئولیت نمونه‌سازی یک شیء از کلاس به کلاس کارخانه را بر عهده می‌گیرد.

[1] Gang of Four
[2] Creational
[3] Structural
[4] Behavioral

۳	کارخانه چکیده (Abstract Factory)	به ما امکان می‌دهد یک کارخانه برای کلاس‌های کارخانه ایجاد کنیم.
۴	سازنده (Builder)	ایجاد یک شیء گام‌به‌گام و روشی برای رسیدن به نمونه شیء (object instance)
۵	نمونه اولیه (Prototype)	ایجاد یک نمونه شیء جدید از یک نمونه مشابه دیگر و سپس اصلاح بر اساس نیازهای ما

جدول ۳ - الگوهای طراحی ساختاری

ردیف	نام الگو	توصیف
۶	آداپتور (Adapter)	یک رابط بین دو موجودیت نامرتبط را فراهم می‌کند تا بتوانند با هم کار کنند.
۷	ترکیب (Composite)	زمانی استفاده می‌شود که باید یک سلسله‌مراتب بخشی از کل را پیاده‌سازی کنیم. به‌عنوان مثال، نموداری که از قطعات دیگری مانند دایره، مربع، مثلث و غیره ساخته شده است.
۸	پروکسی (Proxy)	فراهم‌کردن یک جانشین یا مکان نگهدار برای یک شیء دیگر برای کنترل دسترسی به آن
۹	Flyweight	ذخیره‌سازی و استفاده مجدد از نمونه‌های شی، که با اشیاء تغییرناپذیر استفاده می‌شود. به‌عنوان مثال string pool
۱۰	نما (Facade)	ایجاد واسطهای پوششی در بالای رابطهای موجود برای کمک به برنامه‌های مشتری
۱۱	پل (Bridge)	الگوی طراحی پل برای جداکردن رابطها از پیاده‌سازی و پنهان‌کردن جزئیات پیاده‌سازی از برنامه مشتری استفاده می‌شود.
۱۲	Decorator	الگوی طراحی دکوراتور برای اصلاح عملکرد یک شیء در زمان اجرا استفاده می‌شود.

جدول ۴ - الگوهای طراحی رفتاری

ردیف	نام الگو	توصیف
۱۳	روش قالب (Template Method)	برای ایجاد یک روش خرد الگو و موکول‌کردن برخی از مراحل پیاده‌سازی به زیر کلاس‌ها استفاده می‌شود.
۱۴	واسطه (Mediator)	برای ارائه یک رسانه ارتباطی متمرکز بین اشیاء مختلف در یک سیستم استفاده می‌شود.
۱۵	زنجیره مسئولیت (Chain of Responsibility)	برای دستیابی به اتصال شل[1] در طراحی نرم‌افزار استفاده می‌شود که در آن درخواست مشتری به زنجیره‌ای از اشیاء ارسال می‌شود تا آنها را پردازش کند.
۱۶	مشاهده‌کننده (Observer)	زمانی مفید است که به وضعیت یک شیء علاقه‌مند هستید و می‌خواهید هر زمان که تغییری رخ داد مطلع شوید.

[1] Loose Coupling

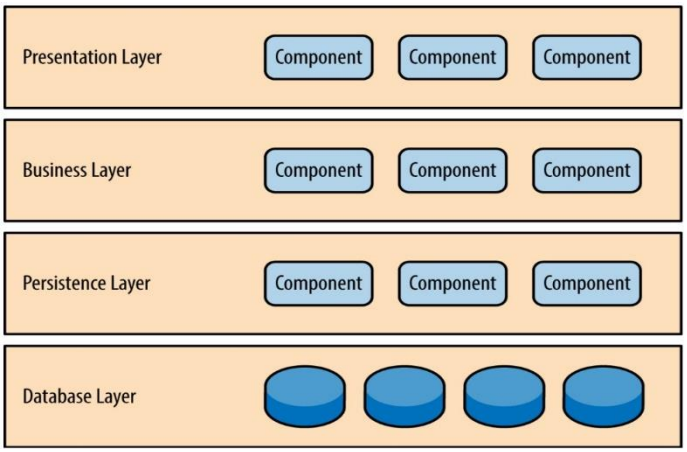
۱۷	استراتژی (Strategy)	زمانی استفاده می‌شود که چندین الگوریتم برای یک کار خاص داشته باشیم و مشتری تصمیم بگیرد که پیاده‌سازی واقعی در زمان اجرا مورد استفاده قرار گیرد.
۱۸	فرمان (Command)	برای پیاده‌سازی اتصال شل در یک مدل درخواست - پاسخ استفاده می‌شود.
۱۹	حالت (State)	زمانی استفاده می‌شود که یک شیء رفتار خود را بر اساس وضعیت داخلی خود تغییر دهد.
۲۰	بازدیدکننده (Visitor)	زمانی استفاده می‌شود که باید عملیاتی را روی گروهی از نوع مشابه از اشیاء انجام دهیم.
۲۱	مفسر (Interpreter)	یک نمایش گرامری برای یک زبان تعریف می‌کند و یک مترجم برای مقابله با این دستور زبان ارائه می‌دهد.
۲۲	اشاره‌گر (Iterator)	برای ارائه یک راه استاندارد برای عبور از میان گروهی از اشیاء استفاده می‌شود.
۲۳	یادگاری (Memento)	الگوی طراحی یادگاری زمانی استفاده می‌شود که می‌خواهیم وضعیت یک شیء را ذخیره کنیم تا بتوانیم بعداً آن را بازیابی کنیم.

۳-۱۲- الگوی لایه‌ای[1]

رایج‌ترین الگوی معماری، الگوی معماری لایه‌ای است که به‌عنوان الگوی معماری n-ردیف[2] شناخته می‌شود. این الگو استاندارد واقعی برای اکثر برنامه‌های کاربردی Java EE است و بنابراین به طور گسترده توسط اکثر معماران، طراحان و توسعه‌دهندگان شناخته شده است. الگوی معماری لایه‌ای با ارتباطات سنتی فناوری اطلاعات و ساختارهای سازمانی که در اکثر شرکت‌ها یافت می‌شود، مطابقت دارد و آن را به یک انتخاب طبیعی برای اکثر تلاش‌های توسعه برنامه‌های تجاری تبدیل می‌کند. (Richards, 2015).

اجزای درون الگوی معماری لایه‌ای در لایه‌های افقی سازمان‌دهی می‌شوند که هر لایه نقش خاصی را در برنامه انجام می‌دهد (به‌عنوان مثال، منطق ارائه[3] یا منطق تجاری[4]). اگرچه الگوی معماری لایه‌ای تعداد و انواع لایه‌هایی را که باید در الگو وجود داشته باشد مشخص نمی‌کند، اکثر معماری‌های لایه‌ای از چهار لایه استاندارد تشکیل شده‌اند: ارائه[5]، کسب‌وکار[6]، تداوم[7] و پایگاه داده[8] (شکل ۱۷) (Richards, 2015).

[1] Layered
[2] n-tier
[3] Presentation logic
[4] Business logic
[5] Presentation
[6] Business
[7] Persistence
[8] Tatabase



شکل ۱۹ – الگوی معماری لایه‌ای

معماری لایه‌ای مسئولیت‌های نرم‌افزار را به چندین لایه به‌هم‌پیوسته تقسیم می‌کند. یک سیستم بر اساس جداسازی نگرانی، لایه‌بندی شده است. لایه‌ها می‌توانند از طریق واسطهای کاملاً تعریف شده با یکدیگر ارتباط برقرار کنند. ازآنجایی‌که لایه‌ها به طور سست جفت شده‌اند، می‌توانند روی یک کامپیوتر قرار بگیرند یا ممکن است روی چندین رایانه توزیع شوند. می‌توان در یک زمان روی لایه‌های مختلف سیستم کار کرد و به‌راحتی سیستم را مقیاس و نگهداری کرد. در اینجا کاربران مستقیماً یک مؤلفه را فراخوانی نمی‌کنند؛ در عوض ایجاد ارتباط از طریق لایه‌ها انجام می‌شود. الگوی معماری لایه‌ای، می‌تواند کل ارتباط بین اجزای مختلف سیستم را کُند کند. همچنین یافتن تعداد دقیق لایه‌ها برای یک سیستم دشوار است (Sharma et al., 2015).

٤- مقایسه الگوهای معماری

با مطالعه و جمع‌آوری مطالب از منابع مختلف، مقایسه خصوصیات کیفی برای هر یک از الگوهای معماری موردبحث، در جداول ۵ و ۶ نشان داده است. در جداول زیر، H مخفف High، M مخفف Medium و L مخفف Low است.

جدول ۵ – بررسی خصوصیات کیفی الگوهای معماری

الگوهای معماری ← / خصوصیات کیفی ↓	Micro Service	Client-Server	MVC	MVP	MVVM	Pipe & Filter
پیچیدگی[1]	M to H	L to M	M	M	M	M
قابلیت اطمینان[2]	M to H	M to H	M	M	M	H
مقیاس‌پذیری[3]	H	M to H	M	M	M	H
عملکرد[4]	H	H	H	H	H	H
بهره‌وری[5]	M to H	M to H	M	M	M	M to H
قابلیت حمل[6]	H	H	H	H	H	H

[1] Complexity
[2] Reliability
[3] Scalability
[4] Functionality
[5] Efficiency
[6] Portability

قابلیت استفاده[1]	M	H	H	H	H	M
قابلیت نگهداری[2]	H	H	H	H	H	H
مثال	Netflix, Amazon, Twitter, LinkedIn	Microsoft Outlook, Gmail, Facebook, Skype	Ruby on Rails, Django, Express.js	Google Web Toolkit, GWTP, Robolectric	Microsoft WPF, Microsoft Silverlight, React Native	Unix/Linux Command Line Tools

جدول ۶ - بررسی خصوصیات کیفی الگوهای معماری

الگوهای معماری ← / خصوصیات کیفی ↓	Domain Driven Design (DDD)	Event-driven	Gang of four (GOF)	Layered
پیچیدگی	M to H	M to H	L to M	L to M
قابلیت اطمینان	M to H	M to H	Depends on Implementation	H
مقیاس‌پذیری	M to H	H	Depends on Implementation	M to H
عملکرد	H	H	Depends on Implementation	H
بهره‌وری	M to H	M to H	Depends on Implementation	M to H
قابلیت حمل	M to H	M to H	H	H
قابلیت استفاده	M to H	M to H	M to H	H
قابلیت نگهداری	H	M to H	H	H
مثال	Customer relationship management (CRM) systems	Amazon S3 (Simple Storage Service)	توضیحات در ادامه آمده است.	توضیحات در ادامه آمده است.

مثال‌های دنیای واقعی از الگوهای GOF:

- الگوی Singleton: اغلب در چارچوب‌های ورود به سیستم، استخرهای اتصال پایگاه‌داده[3]، و استخرهای رشته‌ای[4] استفاده می‌شود، جایی که باید یک نمونه واحد و سراسری از یک کلاس خاص وجود داشته باشد.
- الگوی Observer: به طور گسترده در رابط‌های کاربر گرافیکی (GUI)، سیستم‌های رویدادمحور و سیستم‌های انتشار - اشتراک استفاده می‌شود، جایی که اشیاء نیاز به اطلاع و به‌روزرسانی اشیاء دیگر در مورد تغییرات وضعیت دارند.
- الگوی Factory: در چارچوب‌ها و کتابخانه‌ها به کار می‌رود تا یک رابط برای ایجاد اشیا فراهم کند و به کلاس‌های فرعی اجازه می‌دهد تا نوع دقیق اشیاء را که باید نمونه‌سازی شوند، تصمیم بگیرند.
- الگوی Adapter: الگوی آداپتور اغلب در سناریوهای یکپارچه‌سازی سیستم استفاده می‌شود، جایی که کلاس‌ها یا رابط‌های موجود باید برای کار با رابط‌های جدید یا متفاوت تطبیق داده شوند.

[1] Usability
[2] Maintainability
[3] Database Connection Pools
[4] Thread Pools

- الگوی Proxy: الگوی پروکسی در سیستم‌هایی استفاده می‌شود که از یک شیء جایگزین برای کنترل دسترسی به یک شی دیگر استفاده می‌شود. به‌عنوان مثال می‌توان به پراکسی‌های راه دور در سیستم‌های توزیع شده و بارگذاری تنبل اشیا[1] اشاره کرد.
- الگوی Strategy: الگوی استراتژی معمولاً در سیستم‌هایی استفاده می‌شود که نیاز به الگوریتم‌ها یا رفتارهای قابل تعویض دارند. به‌عنوان مثال می‌توان به الگوریتم‌های مرتب‌سازی در کتابخانه‌ها و استراتژی‌های مختلف قیمت‌گذاری در سیستم‌های تجارت الکترونیک اشاره کرد.

مثال‌های دنیای واقعی از الگوی لایه‌ای:

- سیستم عامل‌ها: سیستم عامل‌هایی مانند ویندوز، macOS و لینوکس با استفاده از معماری لایه‌ای ساختار یافته‌اند. این لایه‌ها شامل هسته، گرداننده‌های دستگاه، سرویس‌های سیستم و رابط‌های کاربری هستند که هر کدام وظایف خاصی را بر عهده دارند.
- برنامه‌های کاربردی وب: بسیاری از برنامه‌های کاربردی وب از الگوی معماری لایه‌ای پیروی می‌کنند. لایه‌ها معمولاً شامل یک لایه ارائه (UI)، لایه منطق تجاری (منطق برنامه) و لایه دسترسی به داده (عملیات پایگاه‌داده) هستند.
- پروتکل‌های شبکه: پروتکل‌هایی مانند مدل TCP/IP و OSI مبتنی بر معماری لایه‌ای هستند. هر لایه مسئول جنبه خاصی از ارتباطات شبکه مانند کپسوله‌کردن داده‌ها، مسیریابی و مدیریت خطا است.
- برنامه‌های کاربردی موبایل: چارچوب‌های توسعه اپلیکیشن موبایل مانند اندروید و iOS اغلب از یک الگوی معماری لایه‌ای پیروی می‌کنند. لایه‌ها ممکن است شامل UI، منطق تجاری، دسترسی به داده‌ها و ارتباطات شبکه باشند.

۵- بحث و نتیجه‌گیری

ارائه نرم‌افزار با کیفیت مطلوب که نیازمندی‌های کسب‌وکار را برآورده کند، هدف اصلی هر پروژه نرم‌افزاری است. فرایند توسعه نرم‌افزار، شامل مراحل مختلف شناسایی نیازمندی‌ها و تحلیل آن‌ها، طراحی، پیاده‌سازی، آزمون و نگهداری نرم‌افزار می‌باشد. هر مرحله از این فرایند دارای اهمیت و فنون خاص خود است که بهبود کیفیت و توسعه مؤثر نرم‌افزار را فراهم می‌کند. در گذشته توسعه سنتی، بر اساس خطوط کد بود. امروزه بر اساس زمان تحویل است. سازمان‌های نرم‌افزاری به‌طور فزاینده‌ای با فشارهای هم‌زمان مواجه می‌شوند:

۱. کاهش زمان ورود به بازار. ۲. کاهش قیمت تمام‌شده محصول. ۳. بهبود بهره‌وری سازمان.
۴. افزایش قابلیت اطمینان محصول. ۵. افزایش کیفیت محصول.

این فشارها به محیط‌های نرم‌افزاری در حال تکامل از محیط‌های توسعه یکپارچه، چارچوب‌ها، ماژول‌های طراحی و سکوهای مبتنی بر وب منبع‌باز ناشی می‌شود. در این میان کلید توسعه نرم‌افزار، چارچوب‌های توسعه‌پذیر و الگوهای طراحی است.
اگرچه تمام مراحل چرخه عمر توسعه نرم‌افزار (SDLC[2]) اهمیت خاص خود را دارند، اما معماری نرم‌افزار به‌عنوان پایه‌ای برای سایر مراحل SDLC عمل می‌کند. معماری نرم‌افزار به ساختار و سازماندهی سیستم‌های نرم‌افزاری می‌پردازد. یک معماری نرم‌افزار که به‌درستی طراحی شده باشد، برای سازمان‌ها و توسعه‌دهندگان امکاناتی ارائه می‌دهد که در توسعه، نگهداری و مدیریت نرم‌افزار مؤثر واقع شوند. در فرایند معماری نرم‌افزار، الگوهای معماری از اهمیت بسیاری برخوردار هستند. انتخاب الگوی معماری

[1] Lazy loading of objects
[2] Software Development Life Cycle (SDLC)

مناسب، نخستین انتخاب مهم معمار در طراحی یک معماری است. در این مقاله، ۱۰ الگوی رایج و متداول معماری نرم‌افزار مورد بررسی قرار گرفت و مزایا و معایب هر یک از آنها شرح داده شد.

انتخاب الگوی معماری مناسب برای نرم‌افزار، نقش حیاتی در موفقیت نرم‌افزار دارد. بااین‌حال، انتخاب نادرست الگوی معماری برای نرم‌افزار می‌تواند منجر به طراحی اشتباه معماری نرم‌افزار شود. طراحی اشتباه معماری منجر به توسعه دیرهنگام، دوباره‌کاری پرهزینه و شکست کامل نرم‌افزار می‌شود. در مقابل، انتخاب درست الگوی معماری می‌تواند بهبود قابلیت توسعه، قابلیت نگهداری و کیفیت نرم‌افزار را به همراه داشته باشد.

مراجع

Abdullah, A., Phamhung, P., & Namhuh, E. (2017). An Architecture of Thin Client in Internet of Things and Efficient Resource Allocation in Cloud for Data Distribution.

Ahlan, A. R., Mahmud, M., & Arshad, Y. (2010). Conceptual architecture design and configuration of thin client system for schools in Malaysia: A pilot project. *Proceedings 2010 International Symposium on Information Technology - Engineering Technology, ITSim'10*, *2*, 952-955. https://doi.org/10.1109/ITSIM.2010.5561585

Anuar, S. M. S., Azmi, N. F. M., Maarop, N., Samy, G. N., Yaacob, S., & Ten, D. W. H. (2017). Preliminary review of model-view-presenter (MVP) and usability design for the development of postgraduate web portal. *Open International Journal of Informatics*, *5*(1), 1-11.

Aychew, M., & Alemneh, E. (2022). Selection of Architectural Patterns based on Tactics. 2022 International Conference on Information and Communication Technology for Development for Africa (ICT4DA),

Babar, M. A., & Gorton, I. (2009). Software architecture review: The state of practice. *Computer*, *42*(7), 26-32.

Bass, L., Clements, P., & Kazman, R. (2021). *Software Architecture in Practice* (4th ed.). Addison-Wesley Professional. https://books.google.com/books/about/Software_Architecture_in_Practice_4th_Ed.html?id=BWpuzgEACAAJ

Bode, S., & Riebisch, M. (2010). Impact evaluation for quality-oriented architectural decisions regarding evolvability. European Conference on Software Architecture,

Braun, S., Bieniusa, A., & Elberzhager, F. (2021). Advanced domain-driven design for consistency in distributed data-intensive systems. Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data,

Fernando, C. (2022). Designing Enterprise Platforms with Event-Driven Architecture Patterns. In *Solution Architecture Patterns for Enterprise: A Guide to Building Enterprise Software Systems* (pp. 147-179). Springer.

Fowler, M. (2012). *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissidess, J. (2015). *Design Patterns: Elements of Reusable Object-Oriented Software - Paperback* (1st ed.). Pearson Education India.

García, R. F. (2023). MVP: Model–View–Presenter. In *iOS Architecture Patterns: MVC, MVP, MVVM, VIPER, and VIP in Swift* (pp. 107-144). Springer.

Garriga, M. (2018). Towards a taxonomy of microservices architectures. Software Engineering and Formal Methods: SEFM 2017 Collocated Workshops: DataMod, FAACS, MSE, CoSim-CPS, and FOCLASA, Trento, Italy, September 4-5, 2017, Revised Selected Papers 15,

Ghofrani, J., & Lübke, D. (2018). Challenges of Microservices Architecture: A Survey on the State of the Practice. *ZEUS*, *2018*, 1-8.

Gilbert, J., & Price, E. (2021). *Software Architecture Patterns for Serverless Systems: Architecting for innovation with events, autonomous services, and micro frontends*. Packt Publishing, Limited.

Harrison, N. B., & Avgeriou, P. (2010). How do architecture patterns and tactics interact? A model and annotation. *Journal of systems and software*, *83*(10), 1735-1758.

Hassan, A., & Oussalah, M. (2018). Evolution Styles: Multi-View/Multi-Level Model for Software Architecture Evolution. *Journal of Software*, *13*. https://doi.org/10.17706/jsw.13.3.146-154

Jacob, P. M., & Mani, P. (2018). Software architecture pattern selection model for Internet of Things based systems. *IET Software*, *12*(5), 390-396.

Jailia, M., Kumar, A., Agarwal, M., & Sinha, I. (2016). Behavior of MVC (Model View Controller) based Web Application developed in PHP and. NET framework. 2016 International Conference on ICT in Business Industry & Government (ICTBIG),

Jaramillo, D., Nguyen, D., & Smart, R. (2016). *Leveraging microservices architecture by using Docker technology*. https://doi.org/10.1109/SECON.2016.7506647

Kassab, M., & El-Boussaidi, G. (2013). Towards Quantifying Quality, Tactics and Architectural Patterns Interactions (S). SEKE,

Kassab, M., El-Boussaidi, G., & Mili, H. (2012). A quantitative evaluation of the impact of architectural patterns on quality requirements. *Software Engineering Research, Management and Applications 2011*, 173-184.

Kassab, M., Mazzara, M., Lee, J., & Succi, G. (2018). Software architectural patterns in practice: an empirical study. *Innovations in Systems and Software Engineering*, *14*, 263-271.

Kleppmann, M., Beresford, A. R., & Svingen, B. (2019). Online event processing. *Communications of the ACM*, *62*(5), 43-49.

Krause, T., Zickfeld, M., Bruchhaus, S., Reis, T., Bornschlegl, M. X., Buono, P., Kramer, M., Mc Kevitt, P., & Hemmje, M. (2023). An Event-Driven Architecture for Genomics-Based Diagnostic Data Processing. *Applied Biosciences*, *2*(2), 292-307.

Kumar, S. (2019). A review on client-server based applications and research opportunity. *International Journal of Recent Scientific Research*, *10*(7), 33857-33386.

Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., & Babar, M. A. (2021). Understanding and addressing

quality attributes of microservices architecture: A Systematic literature review. *Information and Software Technology*, *131*, 106449.

Márquez, G., Astudillo, H., & Kazman, R. (2023). Architectural tactics in software architecture: A systematic mapping study. *Journal of systems and software*, *197*, 111558.

Mondal, A. K., Schneider, K. A., Roy, B., & Roy, C. K. (2022). A survey of software architectural change detection and categorization techniques. *Journal of systems and software*, *194*, 111505.

Nyabuto, G. (2024). Architectural Review of Client-Server Models. *International Journal of Scientific Research and Engineering Trends*, *10*, 139-143.

Olukunle, O., & Oyerinde, I. (2022). A REVIEW ON SOFTWARE ARCHITECTURAL PATTERNS. *9*, 26-30.

Onarcan, O., & Fu, Y. (2018). A Case Study on Design Patterns and Software Defects in Open Source Software. *Journal of Software Engineering and Applications*, *11*, 249-273. https://doi.org/10.4236/jsea.2018.115016

Ortiz Fuentes, J. D., & Herranz Nieva, Á. (2022). Persistence Factories Architectural Design Pattern. European Conference on Software Architecture,

Osman, A., Brückner, P., Salah, H., Fitzek, F., Strufe, T., Fischer, M., & Tu. (2019). *Sandnet: Towards High Quality of Deception in Container-Based Microservice Architectures*. https://doi.org/10.1109/ICC.2019.8761171

Qureshi, M. R., & Sabir, F. (2014). A comparison of model view controller and model view presenter.

Richards, M. (2015). *Software architecture patterns.*

Sabry, A. E. (2015). Decision model for software architectural tactics selection based on quality attributes requirements. *Procedia Computer Science*, *65*, 422-431.

Scott, J., & Kazman, R. (2009). Realizing and refining architectural tactics: Availability.

Sharma, A., Kumar, M., & Agarwal, S. (2015). A complete survey on software architectural styles and patterns. *Procedia Computer Science*, *70*, 16-28.

Sheikh, W., & Sheikh, N. (2020). A model-view-viewmodel (MVVM) application framework for hearing impairment diagnosis-Class inheritance architecture. 2020 Intermountain Engineering, Technology and Computing (IETC),

Shreelekhya, G., Yazhini, S. U., & Manikandan, N. (2016). Methods for evaluating software architecture-A survey. *International Journal of Pharmacy and Technology*, *8*(4), 25720-25733.

Spillner, A., & Linz, T. (2021). *Software Testing Foundations, 5th Edition: A Study Guide for the Certified Tester Exam*. Rocky Nook. https://books.google.com/books?id=6g6MzgEACAAJ

Tune, N., & Millett, S. (2015). *Patterns, Principles, and Practices of Domain-Driven Design*. Wiley.

Velasco-Elizondo, P., Marín-Piña, R., Vazquez-Reyes, S., Mora-Soto, A., & Mejia, J. (2016). Knowledge representation and information extraction for analysing architectural patterns. *Science of Computer Programming*, *121*, 176-189.

Voorhees, D. (2020). SD Case Study: Model–View–Controller. In (pp. 219-247). https://doi.org/10.1007/978-3-030-28501-2_16

Yu, L., Li, Y., & Ramaswamy, S. (2017). Design Patterns and Design Quality: Theoretical Analysis, Empirical Study, and User Experience. *International Journal of Secure Software Engineering*, *8*, 53-81. https://doi.org/10.4018/IJSSE.2017040103

Zakoldaev, D., Gurjanov, A., Shukalov, A., & Zharinov, I. (2019). Client-server technologies at the enterprises of Industry 4.0. *IOP Conference Series: Materials Science and Engineering*, *656*, 012059. https://doi.org/10.1088/1757-899X/656/1/012059

کریمی, ع., طلوعی فر, ع. و پالوایه, ا. (۱۴۰۲). مروری بر الگوهای متداول در فرایند معماری نرم افزار بیستمین کنفرانس بین المللی فناوری اطلاعات، کامپیوتر و مخابرات https://civilica.com/doc/1769202 ,

لطفی, ر., و زمانی, ب. (۱۳۹۴). الگوی *MVC+MVC++* برای توسعه برنامه های کاربردی و تحت وب سومین کنفرانس بین المللی پژوهشهای کاربردی در مهندسی کامپیوتر و فن آوری اطلاعات, تهران https://civilica.com/doc/466770 .

Common Softwares architectural patterns and Styles at a glance

Ali Karimi
Assistant Professor of Imam Hossein Comprehensive University

Vahid Soutdeh[1]
Ph.D. candidate at Imam Hossein Comprehensive University

Hossein Khalili
MSc. Student of Imam Hossein Comprehensive University

1-1- Abstract

To develop a software, different steps must be planned and implemented, which are called the software development process. If the analyst, designer, architect and other people who play a role in the software development team do not pay attention to quality characteristics such as scalability, aggregation, changeability, reliability, etc. from the beginning, the produced software will lack the necessary quality; even if it has implemented all the requirements of duty. Software architecture is one of the vital and effective mechanisms for organizing software systems. In this process, architectural patterns serve as a basis for design to ensure the mentioned quality characteristics. Through optimization of the development and maintenance process, these models improve quality and performance, increase reliability and stability of systems, increase user satisfaction and business efficiency. Knowing the features in them will facilitate the selection of the template(s) suitable for the software being developed and will lead to their correct use.

Considering the importance of this topic in the software development process, this article examines 13 common software architecture patterns and styles in terms of structure, features, advantages and disadvantages and how to use them with the aim of improving the quality and performance of the software will give.

2-1- Keywords: Software architecture, software architecture patterns, quality attributes, quality assurance

1-Corresponding Author: Ali Karimi*