الگوها و سبک های معماری نرم افزار: مروری کلی بر سبک های متداول معماری

کامیار خدامرادی

khodamoradi@ce.sharif.edu

خلاصه

با گسترش روزافزون دامنه تقاضاهای کاربران کامپیوتر و به دنبال آن اندازه سیستم های نرم افزاری، دیگر روش ها و سبک های کلاسیک تولید نرم افزار، پاسخ گوی این نیازمندیها نبوده اند. در دنیای امروز، طراحی یک نرم افزار موفق تنها انتخاب و یا ایجاد ساختمان داده های مناسب و الگوریتم های کارآمد نیست. حجم نرم افزارهای تجاری سالهای اخیر، مهندسان نرم افزار را بر آن داشته که برای غلبه بر پیچیدگی های حاصل از این حجم بالا، به دنبال تکنیک های استفاده مجدد از نرم افزار و روش های Component-based بروند. علی رغم تمام نکات مثبتی که در استفاده از این روش ها وجود دارد، باید به این نکته هم اشاره کرد که مشکلات و مسائل جدیدی نیز به همراه این دید نوین، پا به دنیای نرم افزار گذاشته اند: در یک سیتم نرم افزاری بزرگ که متشکل از اجزای گوناگون خواهد بود، نحوه سازماندهی و ارتباطات این اجزا با یکدیگر چگونه باشد تا نیازمندی های تعیین شده برای نرم افزار برآورده شوند؟ پاسخ به این پرسش، وظیفه اصلی معماران نرم افزار است.

با توجه به اهمیتی که شاخه نسبتاً نوظهور معماری نرم افزار در مهندسی نرم افزار روز دنیا پیدا کرده است، در این مقاله سعی خواهیم کرد روش ها و سبک های موجود و متداول معماری نرم افزار را بررسی کرده، مزایا و معایب هر کدام را بیان کنیم و به این مسئله بپردازیم که هر سبک، مناسب کدام کلاس از نرم افزارهایی است که امروزه تولید و روانه بازار می شوند.

کلمات کلیدی: *معماری نرم افزار، مهندسی نرم افزار، آنالیز و طراحی نرم افزار.*

۱ معرفی سبک های معماری نرم افزار

همانگونه که اشاره شد، در دهه اخیر، نیاز به روش ها و راه حل های نوینی برای طراحی نرم افزارهای بزرگ و پیچیده احساس شده است و این امر به دلیل آنست که روشهای قدیمی و کلاسیک جوابگوی نیازهای روز بازار نرم افزار نبوده اند. در طراحی نرم-افزار به شیوه کلاسیک، سعی می شد دید بالا به پایین[1] به نرم افزار به *طور* مستقیم به اجرا درآید. به این معنی که ابتدا نرم افزار*/سیستم* در بستر محیط هدف[2] (محیطی که نهایتاً نرم افزار می بایست در آن نصب شده و مورد استفاده قرا بگیرد) مورد بررسی قرار گرفته و ارتباطش با اجزای دنیا خارج مشخص می شد (خروجی این مرحله در روشهای ساخت یافته معمولاً نموداری به نام *نمودار* متن است)، سپس نرم افزار به اجزای کوچکتری تقسیم می شد و هر جزء نیز به طور جداگانه مورد بررسی قرار می-

[1] Top-Down
[2] Target Environment

گرفت (Decomposition). علی رغم آنکه دید بالا به پایین به تولید و طراحی نرم افزار همچنان مورد استفاده طراحان این سیستم ها قرار می گیرد، نحوه به کار گیری آن، دستخوش تغییرات اساسی شده است.

تولید نرم افزار که کمک روش های کلاسیک که نمونه ای کلی از آن ذکر شد، در مورد نرم افزارهای بزرگ امروزی بسیار دیر به جواب می رسد (اگر اساساً قادر باشد به جوابی برسد). منظور از یافتن جواب، پیدا کردن شیوه ای برای طراحی نرم افزار تعیین شده، با پارامترهای کیفیت مورد توافق مشتریان و شرکت یا سازمان تولید کننده است. به همین جهت و در جستجوی یک ایده نوین طراحی که بتواند قابلیت تولید[3] بالاتری داشته، پاسخ گوی نیازهای بازار نرم افزاری باشد، متخصصان این امر حاضر به پرداخت بهای چالش های مطرح در معماری نرم افزار شده و به دنبال روشهای مبتنی بر استفاده از اجزای از پیش ساخته رفتند که این خود، بحث معماری نرم افزار را پیش مطرح می کند. موارد زیر نمونه هایی از دلایلی هستند که آشنایی با معماری نرم افزار را برای هر طراح نرم افزاری اجتناب ناپذیر و ضروری می سازند:

- آشنایی با مدل های کلی طراحی نرم افزار، نقش اساسی در درک روابط کلی میان اجزای نرم افزار بازی می کند و طراح را قادر می سازد که سیستم جدید را در قالب مدل بهبود یافته سیستم پیشین ایجاد نماید که این خود می تواند باعث کاهش هزینه های نهایی تولید نرم افزار شود.
- انتخاب مدل مناسب معماری برای یک نرم افزار زمینه ساز موفقیت آن خواهد بود، حال آنکه انتخاب مدل غلط می تواند فاجعه آمیز باشد. بنابر این طراح سیستم می باید انواع مدلها و سبکهای معماری و مشخصات آنها را به خوبی بشناسد تا بر حسب نیازمندیهای نرم افزار در دست تهیه، معماری مناسب آنرا انتخاب نماید.
- شناخت جزئیات هر یک از سبکهای معماری، به معمار سیستم در انتخاب طراحی بهتر از میان گزینه های موجود کمک می کند.

در این مقاله، سعی خواهیم کرد تعاریف مناسب از الگوها و سبک های معماری نرم افزار ارائه کرده، سبک های رایج را به همراه مشخصات، مزایا و معایب هر کدام بیان نماییم. در قسمت بعدی به تعریف مفاهیم *الگو* و سبک معماری پرداخته، تفاوت های آنها را از دید صاحب نظران این رشته بررسی می کنیم و نهایتاً تعریفی نسبتاً جامع از سبک *معماری نرم افزار* ارائه می نماییم.

۱.۱ مقایسه الگو[4] ها و سبک[5] ها

برای آنکه بتوانیم تفاوت مفاهیم الگو و سبک را بیان نماییم، بهتر است ابتدا با تعریف الگو آشنا شویم و سپس به سراغ تفاوت های آن با سبک برویم.

تعریف ۱. به طور کلی، **الگو** را راه حلی برای یک مسئله طراحی، در حضور مسائل دیگر می دانیم.

---

[3] Productivity
[4] Pattern
[5] Style

در توضیح این تعریف، باید به این نکته اشاره کنیم که در دنیای نرم افزار، عمدتاً مسائل موجود مستقل از یکدیگر نبوده و نیازمندیهایی که برای سیستم مطرح می شوند، معمولاً با یکدیگر همبستگی ها و روابطی دارند. به این معنی که در هنگام طراحی یک نرم افزار، به عنوان مثال نمی توانیم بخشی از کل سیستم را به طور مجزا و منفرد[6] و بدون هیچ گونه وابستگی با سایر بخشها یا نیازمندیها در نظر گرفته، طراحی مناسب را برای آن انجام دهیم و سپس آن را به بقیه سیستم اضافه کنیم. چرا که حداقل انتظاری که از سیستم های واقعی می رود آنست که مشخصه های کیفتی از پیش تعیین شده (نظیر زمان پاسخ مطلوب کاربران، امنیت سیستم، ...) در همه قسمتهای سیستم حضور داشته باشند و این امر، طراحی قسمتهای گوناگون را به این مشخصه ها واسبته خواهد ساخت. در نتیجه ناگزیر هستیم در مورد یک مسئله طراحی، وابستگی های آن به سایر مسائل را نیز مد نظر قرار دهیم. با این حال، مجموعه ای از راه حل ها را که در یک زمینه خاص مطرح می-شوند، به تنهایی الگو نمی نامیم. یک راه حل طراحی باید دارای مشخصه هایی باشد تا بتوان آنرا الگو دانست:

- همانطور که از نام آن انتظار می رود، الگو یک راه حل **تکرار شونده** است. به این معنی که زمانی به یک راه حل الگو می گوییم که گونه ای انتزاعی از آن در چندین سیستم مختلف یافت شود.
- الگو، یک راه حل طراحی **موفق** است. بدیهی است که روشهایی را که در طراحی ما را به بن بست و شکست رسانده اند، مجدداً استفاده نخواهیم کرد، زیرا مایل نیستیم فجایع را تکرار کنیم!

تعریف ۲. زبان الگو[7] به مجموعه ای از الگوها و قوانینی برای ساخت و ساماندهی الگوهای جدید از روی الگوهای اولیه گفته می شود.

در دنیای یک زبان الگو، الگوها نقش مجموعه لغات زبان را بازی می کنند، در حالیکه قوانین گرامر زبان را تشکیل می-دهند که مشخص می سازند یک الگوی جدید به چه نحوی باید ساخته شود، و همچنین در برخورد با یک الگوی جدید و ناشناخته ما را قادر می سازند که تشخیص دهیم آیا این الگو، یک الگوی معتبر این زبان هست یا خیر. به کمک این تعریف می توانیم خط مرزی میان الگو و سبک ترسیم کنیم.

سبکهای طراحی در حقیقت راه حل نیستند، بلکه چارچوبی برای راه حل ها مشخص می کنند. منظور از اینکه چارچوب راه حل را تعیین می کنند این است که با انتخاب سبک، فضای راه حلهای[8] مسئله تقلیل می یابد. به این ترتیب جستجو در این فضا برای یافتن راه حل مناسب ساده تر می شود و به این ترتیب، شانس ما برای پیدا کردن روش بهتر طراحی، افزوده می گردد. با این دید، **سبک ها بیشتر به زبان های الگو شباهت دارند تا به الگوها**. ذکر این نکته ضروری است که خط مرز مشخص شده میان الگو و سبک بسیار ظریف و گاه شکننده است. بدان معنی که در بسیاری از مواقع بر سر آنکه یک روش طراحی را می باید سبک نامید و یا الگو توافق کلی وجود ندارد. امّا برای آنکه درگیر این ظرافت ها نشویم، شباهت ها و تفاوت های الگو و سبک را به این صورت خلاصه می کنیم:

---

[6] Isolated
[7] Pattern Language
[8] Solution Space

الگو و سبک در طراحی از آن جهت به هم شباهت دارند که هر دو با این فلسفه بوجود آمده اند که طراح را در انتخاب راه حل مناسب یاری دهند و به همین منظور، هر کدام راه کارها و دستورالعمل هایی را تجویز می کنند. اما الگو و سبک عمدتاً در سطح درشت دانگی با یکدیگر تفاوت دارند. الگو معمولاً جزئی و دقیق تر یک راه حل برای مسئله طراحی پیشنهاد می کند. به عنوان مثال اگر گفته شود در یک سیستم از الگوی طراحی Bridge استفاده شده است، هر چند به صورت کلی، این دید را بدست می آوریم که پیاده سازی یک نوع داده ای انتزاعی، در قالب یک کلاس واسط[9] و یک کلاس پیاده ساز[10] انجام شده است. همچنین می دانیم که این کار برای جدا سازی پیاده سازی از واسط صورت گرفته تا در صورت تغییر در پیاده سازی، با ثابت نگاه داشتن کلاس واسط، برنامه های استفاده کننده از واسط دستخوش تغییر نشده و تغییرات در سیستم منتشر نشوند. به علاوه به این نکته واقف خواهیم بود که این کار هزینه ای در زمان اجرا به ما تحمیل خواهد کرد، در نامگذاری فایل ها و کلاسها می باید قوانینی را رعایت کنیم، و بسیاری موارد دیگر [۵]. پس همانطور که ملاحظه شد، یک نام ساده می تواند اطلاعات جزئی زیادی، حتی تا سطح پیاده سازی در اختیار ما قرار دهد. اما زمانی که از سبک صحبت می کنیم، همچنان تنها نام سبک اطلاعات مفیدی در مورد مشخصه های سیستم بدست خواهیم آورد، امّا این بار اطلاعات چندان جزئی نخواهند بود. به عنوان مثال زمانی که می گوییم سبک معماری یک نرم افزار، Pipes-and-Filters است، همانطور که در ادامه اشاره خواهد شد، در مورد اینکه اجزای نرم افزار چه ویژگی هایی خواهند داشت، نحوه ارتباط میان این اجزا چگونه خواهد بود، انتقال کنترل اجرا در سطح نرم افزار به چه شکل صورت خواهد پذیرفت، و مواردی از این قبیل دید واضحی بدست می آوریم، اما هیچ کدام از این موارد به دقت و ریزدانگی مواردی که در مورد الگو ذکر شد، نیستند.

حال که توانستیم تا حدی مشخص کنیم چه مفاهیمی در اصل سبک نیستند، وقت آنست که به این سوال پاسخ دهیم که چه مفاهیمی سبک هستند؟ در بخش بعدی تعاریف سبک معماری نرم افزار را از نگاه صاحب نظران این رشته به همراه یک جمع بندی در انتها خواهیم دید.

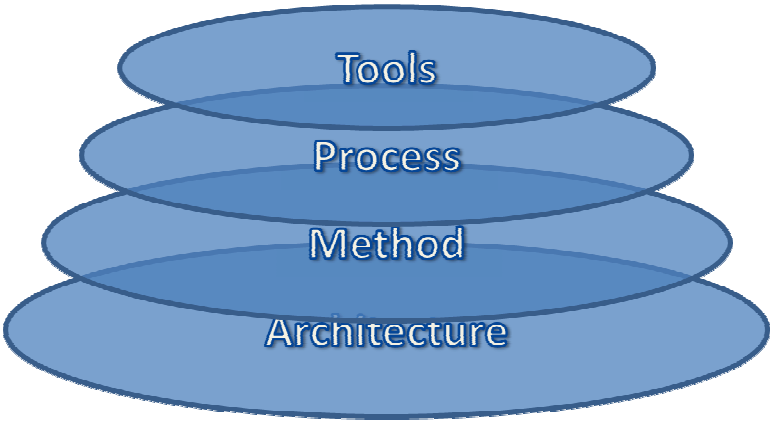
۱.۲ تعریف سبک های معماری نرم افزار

شاو و گارلان، دو تن از صاحب نظران برجسته زمینه معماری نرم افزار، تعریف سبک معماری را این گونه بیان می کنند: *"سبک معماری نرم افزار، مجموعه واژگانی متشکل از اجزای*[11] *نرم افزار و نوع اتصالات میان این اجزا را به همراه مجموعه قوانینی برای ترکیب اجزا و اتصالات با یکدیگر تعریف می کند"*[۳]. با دقت در این تعریف می توان به شباهت سبک معماری نرم افزار از دید شاو و گارلان با زبانهای الگو پی برد. اگر مبنا را تعریف شاو و گارلان قرار دهیم خواهیم دید که بسیاری از نرم افزارها به ویژه نرم افزارهای تجاری، همگن نبوده و چندین سبک مجزا در طراحی آنها به کار رفته است که این خود مسائل جدیدی را مطرح می کند. استفاده از سبکهای مختلف در بخشهای مختلف یک سیستم تنها در صورتی مجاز است که سازگاری میان سبکها حفظ شود. در این مقاله کوتاه نمی توان به طور دقیق به بحث سازگاری سبکها

[9] Interface Class
[10] Implementation Class
[11] Components

پرداخت، با این حال سعی خواهیم کرد در معرفی سبکهای متداول معماری با بیان شباهت ها و/یا تفاوت های آنها به سازگاری هم اشاره ای داشته باشیم.

در یک تعریف دیگر از سبکهای معماری، جِیکوبسن این چنین بیان می کند: "*سبک معماری یک سیستم، نمایش زبان مدلسازی است که در طی فرایند مدل کردن سیستم مورد استفاده قرار گرفته است*"[۲]. در واقع جِیکوبسن معماری نرم-افزار و سبک معماری را با کل فرآیند مدلسازی سیستم معادل می داند و یک ساختار لایه ای هم برای آن ترسیم می کند (شکل ۱).



شکل۱. ساختار لایه ای نرم افزار از دید جِیکوبسن

در این ساختار، معماری نرم افزار و سبکهای آن به عنوان پایه اصلی فلسفه سیستم مطرح هستند. همانگونه که در معماری یک ساختمان، معماری سنگ بنایی برای همه تکنیک ها و روشهای آتی است، در نرم افزار هم به عقیده جِیکوبسن، معماری چشم اندازی به ماهیت اصلی و ذات سیستم در حال توسعه ارائه می دهد. بر روی این لایه روشها قرار دارند که رویه های جزئی تری هستند که وظیفه دارند معماری نرم افزار را پیاده سازی نمایند. در لایه بعدی، پروسه یا فرآیند قرار دارند که دو عنصر **زمان** و **افراد** را به مجموعه روشها اضافه می کند تا کار ساخت سیستم، به صورت پروژه ای قابل برنامه –ریزی و پیگیری در طول زمان درآید. در نهایت و بر فراز همه این لایه ها، ابزار قرار دارند که معماری، روشها، و فرآیند ساخت را یاری می رسانند.

نهایتاً، ریچارد هوبرت سبکهای معماری نرم افزار را به عنوان بخشی از معماری در تعریف خود از معماری نرم افزار می آورد: "*معماری نرم افزار از ۴ قسمت اصلی تشکیل شده است: ۱. متامدل معماری ۲. چرخه حیات توسعه سیتم ۳. ابزارها ۴. Formal Technology Projection* ". در این تعریف متامدل معماری، تقریباً معادل سبک معماری نرم افزار در تعریف جِیکوبسن است و وظیفه آن تعیین اصول و خط مشی اصلی ساخت سیستم نرم افزاری است. چرخه حیات توسعه سیستم مشخص می کند که این خط مشی در عمل به چه طریقی به اجرا درآید و ابزارها هم کمک می کنند که هر کدام از مراحل دیگر انتظاراتی که از آنها می رود را برآورده سازند. مورد آخر را می توان به عنوان گسترشی از ابزارها به حساب آورد. به بیان کلی Formal Technology Projection مجموعه روشهایی است که در طی آن، یک مدل نظیر مدلهای تهیه شده با UML به یک تکنولوژی، مثل دستورات ماشین و یا دستورات یک زبان برنامه نویسی نظیر زبان C نگاشته میشوند.

این روشها برای پشتیبانی از معماری های مدل-گرا[12] بوجود آمده اند و این امکان را فراهم می سازند که همانگونه که یک زبان سطح بالا به مجموعه ای از دستورات زبان سطح ماشین ترجمه می شود، یک مدل سطح بالای طراحی به مجموعه ای از میان افزار[13]ها کامپایل شود. علاوه بر اینکه بسیاری از فعالیت ها در زمینه های فرمال از دید علوم کامپیوتر و تئوری جذابیتهای زیادی دارند، روشهای خودکار تولید نرم افزار در شاخه مهندسی نیز (بر اثر پیچیدگی های روزافزون و فراوان نرم افزارهای مورد تقاضای بازار) روز به روز اهمیت بیشتری می یابند و توجه بیشتری را به خود جلب می کنند.

تعریف ۳. یک سبک معماری نرم افزار مجموعه ای است متشکل از:

- **اجزا**: عناصر اولیه هر نرم افزاری را تشکیل می دهند، نظیر پایگاه های داده، عناصر محاسباتی، و ...
- **اتصالات**: توپولوژی و نحوه ساماندهی و ارتباط اجزا با یکدیگر را نشان می دهند.
- **قوانین**: مجموعه ای از محدودیت ها و مقررات که نحوه ترکیب اجزا، محدودیت های اعمال شده روی اتصالات، و ... را مشخص می کنند.
- **مکانیزم تعامل**: روشی که نشان می دهد اجزای نرم افزار از طریق توپولوژی تعیین شده چگونه با یکدیگر همکاری داشته و کنترل اجرا چگونه میان آنها منتقل می شود. به عنوان مثال فراخوانی ها یک نمونه از این چنین مکانیزمی هستند.

برخی از منابع در رابطه با مورد آخر، مفهومی را با عنوان **مدل فعال سازی**[14] مطرح می کنند که معادل مکانیزم تعامل است. منظور از مدل فعال سازی اینست که:

- اولاً اجزا به چه نحو و در چه زمان هایی برای پردازش اطلاعات فعال می شوند. به زمان هایی که در طی آنها یک المان نرم افزاری فعال است، **زمان فعالیت** می گوییم.
- اطلاعات به چه ترتیبی میان اجزا جابجا می شود.

به طور خلاصه مدل فعال سازی طریقه اجرای منطق اجزای مختلف نرم افزار، شیوه تبادل اطلاعات و همکاری میان آنها، و جریان کنترل میان این اجزا را مشخص می کند.

در پایان این بخش لازم است متذکر شویم که در طبقه بندی سبک های مختلف معماری که در ادامه معرفی خواهیم کرد همپوشانی و اشتراک وجود دارد. با این حال این طبقه بندی به ما کمک می کند که اطلاعات فراوانی را در قالب یک نام خلاصه کنیم. همانطور که در مورد الگوهای طراحی اشاره کردیم که از نام یک الگو می توان دید کلی از آنچه در دست پیاده سازی است پیدا کرد، در مورد سبک های معماری هم طیف وسیعی از افراد زینفع در یک پروژه شامل، مشتریان، تحلیلگران و طراحان سیستم، مدیریت شرکت یا سازمان تولید کنند نرم افزار، کد نویسان، و ... با دانستن سبک معماری یک نرم افزار قادر هستند کلیات و مشخصات کلیدی محصول نهایی را درک کنند، و این امر می تواند کمک بزرگی به شفاف سازی ارتباطات افراد دخیل در پروژه، و نهایتاً موفقیت نرم افزار تولیدی در برآورده ساختن انتظارات بنماید.

---

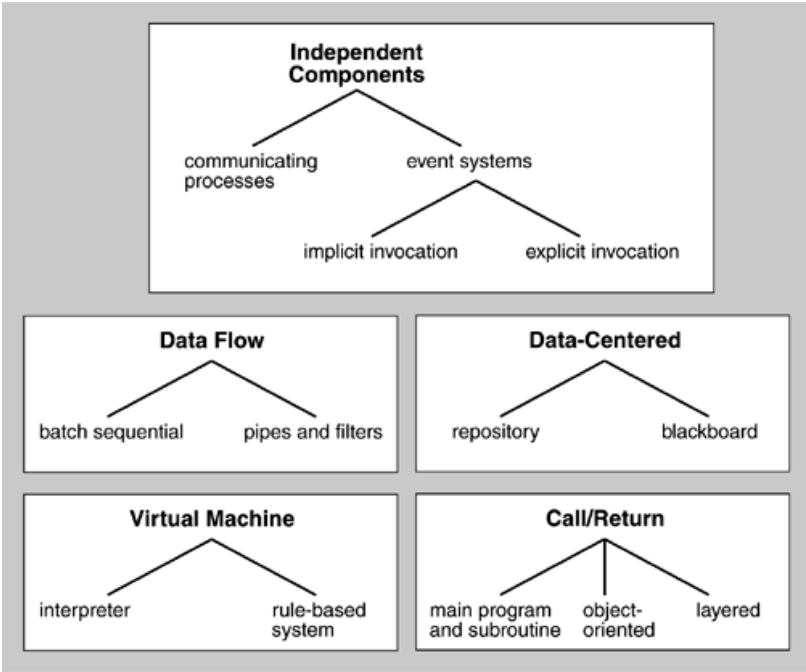
[12] Model-Driven Architecture (MDA)
[13] Middleware
[14] Activation Model

در قسمت بعدی، طبقه بندی شاو و گارلان از سبک ها را بطور کلی بیان کرده و سپس به معرفی سبکهای زیر می پردازیم:

- سبک Implicit Invocation
- سبک Communicating Processes
- سبک Pipes-and-Filters
- سبک Repository
- سبک Blackboard
- سبک Interpreter
- سبک Layered
- سبک Object-Oriented
- سبک Main Program and Sub-routine

۲ سبکهای متدوال معماری نرم افزار

برای نخستین بار در سال ۱۹۹۴ ماری شاو و دیوید گارلان از دانشگاه کارنگی ملون آمریکا دست به طبقه بندی مدون سبک –های معماری زدند که این طبقه بندی امروزه هم به عنوان مرجع در معماری نرم افزار مورد استفاده قرار می گیرد. هر چند این طبقه بندی بسیار خلاصه است و برخی سبک ها را در بر نمی گیرد، امّا پس از گذشت یک دهه همچنان سبک های رایج معماری در نرم افزارهای کاربردی را بخوبی بیان می کند. شکل زیر که بیانگر این طبقه بندی است برگرفته از [۱] است:



شکل ۲. طبقه بندی شاو و گارلان از سبک های معماری نرم افزار

در ادامه مواردی از این طبقه بندی را مختصراً توضیح می دهیم.

۲.۱ سبکهای اجزای مستقل (Independent Component)

این خانواده از سبکهای معماری که شامل دو زیرشاخه Communicating Processes و Event-Based Systems می باشد، عمدتاً بر مبنای احضار ضمنی[۱۵] استوار می باشند، و هر چند احضار صریح[۱۶] هم به عنوان زیرشاخه ای از سیستمهای Event-Based در طبقه بندی آمده است در عمل توجه چندانی به عنوان یک سبک معماری به آن نشده و ما نیز به بحث پیرامون این شیوه (کلمه کلی تر شیوه را بکار می بریم تا وارد جزئیات اینکه آیا احضار صریح یک سبک هست یا نه نشویم) نخواهیم پرداخت.

در این سبک ها تلاش شده است فراخوانی عملیات[۱۷] از اجرای آنها به نحوی جدا شود. به این معنی که درخواست کننده یک سرویس کاملاً مستقل از سرویس دهنده بوده و عمدتاً از وجود آن هم اطلاعی ندارد. حتّی ممکن است در یک سیستم توزیع شده این دو جزء در دو پردازنده مجزا در حال اجرا باشند. البته می باید میان این سبک ها و روشهایی نظیر Remote Procedure Call (RPC) یا معادل شئ گرای آن Remote Method Invocation (RMI) تفاوت قائل شد، زیرا در این دو روش اخیر همچنان فراخوانی سرویس ها یا متدها به صورت صریح صورت می گیرد.

۲.۲.۱ Implicit Invocation Event-Based Systems

در این سبک معماری دو رکن اساسی وجود دارد:

- عملیات
- رویدادها[۱۸]

هر المان نرم افزاری هر دوی این موارد را با هم شامل می شود و در واقع تعدادی عملیات را (به عنوان عکس العمل) به تعدادی رویداد نسبت می دهد. ایده اصلی در پشت احضار ضمنی آنست که به جای آنکه یک عنصر نرم افزاری به طور مستقیم یک متد را فراخوانی کند، یک رویداد ایجاد کند خبر وقوع آن که در کل سیستم پخش همگانی[۱۹] شود. همچنین هر عنصری به وقوع تعدادی از این رویدادها حساس است و به آن رویداد های مشخص، عملیاتی را نسبت داده است. به محض آنکه یک عنصر متوجه وقوع یکی از این رویدادها شود، عملیات تعیین شده و متناظر با آن را به اجرا در می آورد. به این ترتیب یک المان نرم افزاری با تولید یک رویداد به *طور ضمنی* باعث اجرای تعدادی عملیات می شود. نکته در این جاست که به دلیل عدم آگاهی تولید کننده رویداد از پاسخ دهنده یا پاسخ دهندگان به آن، یک پردازه[۲۰] خاص، نمی تواند پیش بینی کند که پردازه های بعدی به چه ترتیبی اجرا می شوند و یا حتّی اینکه چه پردازه هایی ممکن است به اجرا

---

[15] Implicit Invocation
[16] Explicit Invocation
[17] Operations
[18] Events
[19] Broadcast
[20] Process

درآیند. به این دلیل گاهی به سیستم های مبتنی بر احضار ضمنی، امکان احضار صریح را هم به عنوان مکملی برای مدل تعامل اجزای نرم افزار اضافه می کنند. به این ترتیب مدل فعال سازی کاملاً بستگی به آن دارد که اجزای مختلف به چه زیرمجموعه ای از رویدادها علاقه (حساسیت) نشان دهند و عکس العمل ها چه جریان کنترلی را ایجاب نماید.

به عنوان نمونه ای از استفاده از این سبک در نرم افزارها می توان به Trigger ها اشاره کرد. در اثر یک رویداد در سیستم نظیر بروز رسانی اطلاعات ممکن است یک یا چند Trigger فعال شوند و هر کدام عملیاتی را در قبال رویدادی که حس کرده اند، انجام دهند. البته خود این عملیات می تواند سبب ساز فعال سازی Trigger های دیگری بشود. در هر صورت المانی که در ابتدا رویداد را آغاز کرده است، ممکن است اصلاً از وجود Trigger ها اطلاعی نداشته باشد.

۲.۲.۲ سبک Communicating Processes

در این سبک معماری، اصول و قواعد کلی Implicit Invocation Event-Based Systems تا حدودی پابرجاست. با تفاوت که در این سبک تأکید اصلی بر روی تبادل پیغام[۲۱] میان المان های نرم افزار است. این ویژگی سبب می شود که مفاهیمی مانند وقوع رویداد و پخش همگانی رخداد معنی مشخصی به خود بگیرند: وقوع رخداد یعنی رسیدن پیغام به یک عنصر نرم افزاری، و برای پخش همگانی خبر وقوع آن هم کافیست از طریق یکی از پروتکل های ارتباطی پیغام میان سایر اجزا منتشر شود.

استفاده از پیغام می تواند مشکلاتی هم در بر داشته باشد. در این قسمت به یکی از این مسائل اشاره خواهیم کرد و در بخش بعدی مزایا و معایب خانواده سبک های اجزای مستقل را به طور کلی بررسی می کنیم. مشکلی که در عمل در پیاده –سازی بسیاری از سیستم ها بر اساس سبک Communicating Processes بوجود آمده است مسئله جامعیت[۲۲] نرم-افزار است. به این معنی که در سیستم های قدیمی تر عمدتاً مکانیزمی برای تبادل پیغام پیش بینی نشده است و این سیستم ها اتصال گرا[۲۳] طراحی و ساخته شده اند. در نتیجه زمانی که قصد در گسترش آنها بر طبق سبک نوین داریم می –باید به نحوی به مسئله جامعیت سیستم بپردازیم. به این منظور نیاز به یک مبدل یا مترجم داریم که پیغام های مبادله شده در سیستم را به عنوان عامل هایی از یک محیط اتصال گرا به سیستم بشناساند و به این ترتیب کل سیستم را یکپارچه سازد، که البته بدیهی است قرار دادن چنین واسطی میان بخش های مبتنی بر تبادل پیغام و سایر بخشها، می-تواند کارایی کل سیستم را به میزان قابل توجهی کاهش دهد.

۲.۲.۳ مزایا و معایب سبک اجزای مستقل

- **مزایا**

[21] Message Passing
[22] Integrity
[23] Connection-oriented

- **استفاده مجدد از نرم افزار**[۲۴]: اصلی ترین مزیت این سبک پشتیبانی قوی از استفاده مجدد از نرم افزار است. به دلیل عدم آگاهی اجزای مختلف نرم افزار از یکدیگر، وابستگی میان آنها به حداقل می رسد. در نتیجه اضافه کردن یک عنصر جدید به سیستم به سادگی انجام پذیر بوده و تنها کاری که باید انجام داد آنست که تعیین کنیم این عنصر جدید به چه رویدادهایی علاقه مند است.
- **قابلیت گسترش**: به دلیل Modularity بالای نرم افزارهای تهیه شده براساس این سبک، گسترش نرم افزار به سهولت انجام می پذیرد.

- **معایب**

- **کارایی**: به دلیل ارتباط میان پردازه ای[۲۵] هزینه ای از کارایی سیستم باید پرداخت کنیم که این تا حد زیادی به نوع ارتباطات اجزا با هم بستگی دارد.
  - اگر ارتباطات از نوع *ناهمگام* باشند، میزان برقراری ارتباط المانها با یکدیگر به نسبت کمتر بوده و در هر بار برقراری ارتباط حجم بیشتری اطلاعات منتقل می شود. به این ترتیب کارایی افت چشم گیری نخواهد داشت.
  - اگر سیستمی که از این سبک استفاده می کند، نیازمند ارتباطات *همگام* باشد، نظیر سیستم های محاوره ای[۲۶]، در آن صورت تعداد دفعات برقراری ارتباط زیاد، و در هر بار برقراری ارتباط حجم اطلاعات منتقل شده کم خواهد بود و این امر سبب کاهش محسوس کارایی کل سیستم خواهد شد.
- **پیچیدگی**: استفاده از این سبک باعث پیچیده تر شدن نرم افزار هم می شود. به عنوان نمونه، عمل اشکال-زدایی در این نرم افزارها می تواند مشکل و زمانگیر باشد، به این دلیل که همانطور که گفته شد، جریان کنترل میان المانها به وضوح مشخص نیست و نمی توان به راحتی تعیین کرد ایرادی که در حین اجرای یک متد رخ داده است، توسط چه بخشی از سیستم آغاز شده است.

۲.۲ سبکهای جریان داده (Data Flow)

این سبک از معماری، شامل دو زیرشاخه اصلی است: *لوله ها* و *فیلترها* و *پردازش دسته ای* که از این دو مورد تنها به مورد اول می پردازیم. سیستم *های جریان داده* سیستم هایی هستند که نحوه تبادل داده میان اجرای مختلف تعیین کننده مشخصات اصلی آنهاست. به بیان دیگر شیوه جریان داده در سیستم نقش تعیین کننده را در رفتار سیستم بازی می کند. جریان داده در این سیستم ها شباهت زیادی به نحوه اجرای منطق زبان های برنامه نویسی دارد و معمولاً سیستم های جریان داده برای مدل کردن هر نوع جریان کاری[۲۷] می توانند گزینه خوبی محسوب شوند. در این سیستم ها وجود حداقل دو المان که میان آنها داده به جریان درآید الزامی است. پردازش عمدتاً در آنها به صورت ترتیبی انجام می شود، به این

[24] Software Reuse
[25] Inter-Process Communication
[26] Interactive Systems
[27] Work Flow

معنی که خروجی یک عنصر، ورودی عنصر/عناصر بعدی در مسیر جریان داده خواهد بود. در بخش بعدی، عمده ترین زیرشاخه این سبک یعنی سبک لوله ها و فیلترها را معرفی می کنیم.

۲.۲.۱ سبک لوله ها و فیلترها

اجزای اصلی این سبک عبارتند از:

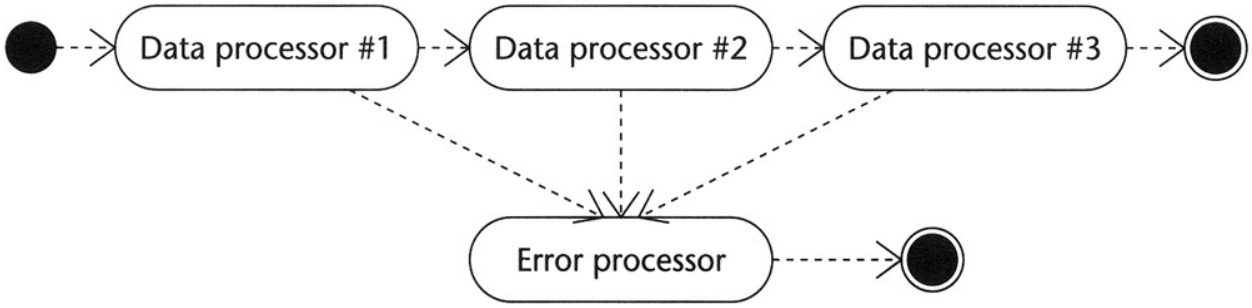
- **فیلترها**: عناصری که وظیفه پردازش داده های ورودی و تبدیل آنها را به اطلاعات خروجی بر عهده دارند.
- **لوله ها**: برقرار کننده ارتباط میان فیلترها هستند و داده ها و اطلاعات را جابجا می کنند.
- **قوانین و محدودیتها**: بر روی نوع لوله ها، ظرفیت آنها، نحوه ترکیب فیلترها با یکدیگر، و ... که در ادامه به برخی از این موارد هم اشاره ای خواهیم داشت.

به هر ورودی یا خروجی به یک فیلتر در این سبک، یک درگاه[۲۸] گفته می شود. فیلترهای استفاده شده در این سبک عمدتاً ۲ درگاه دارند (شکل ۳).

شکل ۳. شمای کلی سبک لوله ها و فیلترها با فیلترهای ۲ درگاهی

تعریف ۴. به هر مسیر جریان داده در یک نرم افزار مبتنی بر سبک لوله ها و فیلترها یک **خط لوله** گفته می شود.

یک مدل ساده از این سبک، این است که تنها یک مسیر برای جریان داده یا تنها یک خط لوله وجود داشته باشد. شکل ۳ نمونه ای از خط لوله با فیلترهای ۲ درگاهی را نشان می دهد. البته نسخه دیگری از فیلترها که ۳ درگاه دارند، بسیار متداول تر 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 خطاهاست، متصل می کنند. شکل ۴ شمای کلی چنین سیستمی را نمایش میدهد.



شکل ۴. شمای کلی سبک لوله ها و فیلترها با فیلترهای ۳ درگاهی – درگاه سوم برای گزارش خطا به فیلتر پردازش کننده خطا تعبیه شده است.

[28] Port

از ویژگی های 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 می گردد. یعنی هر المان پردازشگر مانند فیلتری عمل می کند که در مسیر یک سیالِ در جریان قرار گرفته و تبدیلاتی بر روی آن انجام می دهد. به همین دلیل نام فیلتر بر روی المانهای پردازشگر نهاده شده است. فیلترها معمولاً بر روی هم اثری نمی گذارند، مگر از طریق داده ای که یک فیلتر بالایی[۲۹] برای یک فیلتر پایینی[۳۰] ارسال می نماید. به همین جهت عمدتاً در منابع ذکر شده که فیلترها، حالات خود را به اشتراک نمی گذارند. علاوه بر این یک فیلتر ممکن اساساً از وجود فیلترهای دیگر بی اطلاع باشد (نظیر آنچه که در مورد سبک اجزای مستقل گفته شد). به این ترتیب هر فیلتر تنها لازم است بداند که انتظار دیدن چه نوع ورودی را دارد، و تا زمانی که ورودی به فرمت توافق شده به آن می رسد، تضمیین کند که خروجی تولیدی چه فرمتی داشته باشد. از آنچه تا کنون گفته شد کاملاً روشن است که مدل فعال سازی در این سبک، یک **مدل ترتیبی** است.

قبل از آنکه به مزایا و معیب این سبک بپردازیم، جا دارد برخی ویرایش های صورت گرفته، محدودیت ها، قوانین اعمال شده در این سبک را معرفی کنیم. در این قسمت به انواع لوله هایی که ممکن است در این سبک بکار روند می پردازیم:

تعریف ۵. لوله های نوع دار: به لوله های استفاده شده در این سبک می توان نوع اختصاص داد. متداول ترین انواع لوله عبارتند از:

- **لوله جریانی**[۳۱]: گونه ای از لوله ها که هر جریان از بیت های دودویی را انتقال می دهد. فیترهایی که از این لوله –ها داده دریافت می کنند لازم است ابتدا بر روی داده های دریافتی پیش پردازش انجام دهند (داده ها سریال هستند و باید تبدیل به اشیا قابل تشخیصی نظیر اعداد صحیح، رشته های حرفی، آرایه ها و ... شوند). همچنین فیلترهایی که خروجی خود را روی این نوع لوله ها قرار می دهند، می بایست ابتدا این خروجی را به فرمت سریال درآوند.
- **لوله شئی**[۳۲]: لوله ای را گویند که در آن اشیاء منتقل می شوند. در اینجا منظور از شئ کلی تر از آنست که در بحث شئ گرایی می شناسیم و به معنای هر دسته بندی داده هاست که دربرگیرنده یک مفهوم مشخص (و انتزاعی) باشد.

در لوله های نوع دوم، یک شئ بصورت کلی در فیلتر مصرف شده و یا روی یک خط لوله جابجا می شود و امکان شکستن آن به اشیا کوچکتر وجود ندارد. در استفاده از انواع لوله ها باید دقت داشت که بکارگیری بیش از حد یا نابجای هر کدام می تواند به کارایی کلی سیستم لطمه وارد کند. به عنوان نمونه استفاده محض از لوله های شئی، سبب می شود سیستم به یک سیتم پردازش دسته ای تبدیل شود و دیگر جریان داده ها که زمینه ساز اصلی ویژگی های خوب این سبک نرم افزارهاست از بین برود.

---

[29] Upstream
[30] Downstream
[31] Stream-typed Pipe
[32] Object-typed Pipe

تعریف ۶. لوله های با ظرفیت محدود: بعضی اوقات و به دلیل ملاحظات و نیازمندی های خاص نرم افزار بر روی حجمی از اطلاعات که یک لوله می تواند حمل کند، محدودیت عمال می شود. این کار به عنوان مثال می تواند به دلیل محدودیت پهنای باند در یک سیستمی صورت بپذیرد که بخشی از پردازشها روی شبکه انجام می شود، و فیلترها روی پردازنده های مجزا توزیع شده اند.

به عنوان مثالی از این سبک، می توان به کامپایلرها اشاره کرد که در آنها مراحل pipe line شامل تحلیل لغوی، parsing، تحلیل معنایی و تولید کد می باشد. البته می دانیم که در عمل کامپایلرها عمل ترجمه را به صورت ترتیبی انجام نمی دهند، اما دید کلاسیکی که به کامپایلرها وجود داشته بسیار به این سبک نزدیک است.

۲.۲.۲ مزایا و معایب سبک لوله ها و فیلترها

- **مزایا**
  - در این سبک رفتار کل سیستم به صوت ترکیب ساده ای از رفتار فیلترها مدل می شود.
  - به دلیل استقلال فیلترها از هم، توسع سیستم با سهولت انجام می شود.
  - استفاده مجدد از نرم افزار توسط این سبک پشتیبانی می شود.
  - سبک لوله ها و فیلترها به طور طبیعی از اجرای موازی کارها پشتیبانی می کنند. برای موازی سازی یک عمل، باید ابتدا قسمت هایی 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**
  - به طور ذاتی این سبک تمایل دارد به سمت سیستم های پردازش دسته ای سوق پیدا کند و همانطور که قبلاً گفته شد، با از میان رفتن مفهوم جریان داده در نرم افزار (امکان تولید خروجی قبل از مصرف کامل تولید ورودی) افت کارایی قابل پیش بینی است.
  - در صورت استفاده بیش از حد از لوله های جریانی هم افت کارایی خواهیم داشت. هر بار که یک جریان داده دودویی به یک فیلتر می رسد، آن فیلتر می بایست ابتدا داده ها را پیش پردازش کند تا بتواند بر روی آنها پردازش اصلی را صورت دهد و این کار (Redundant Parsing) سبب افت کارایی خواهد شد.
  - به دلیل آنکه نرم افزار هایی که با این سبک ساخته می شوند ذاتاً به تبدیلات متنوع و مکرر داده ها بستگی دارند، معمولاً این سبک برای نرم افزارهای محاوره ای انتخاب مناسبی نیست.

۲.۳ سبکهای داده-محور (Data Centered)

این خانواده سبک شامل دو زیر شاخه اصلی است که هر کدام را توضیح می دهیم:

۲.۳.۱ سبک Repository

امروزه اکثر شرکت ها و سازمان های بزرگ دنیا وابستگی شدیدی به داده هایشان دارند. حفظ داده های یک شرکت برای ادامه بقای آن به اندازه ای حیاتی است که کمپانی های بزرگ حاضرند میلیون ها دلار صرف امنیت و نگهداری آنها کنند. در فضایی که تا این حد به حفظ داده ها اهمیت می دهد، بروز سبک های معماری نرم افزار مبتنی بر داده های مانا[۳۳] تعجب برانگیز نخواهد بود. با این انگیزه قوی، سبکی از معماری با نام Repository ظهور کرده است که مبنا را برای تبادل اطلاعات میان اجزا، افراد، و ارگانهای مختلف، به اشتراک گذاری داده ها قرار می دهد. اجزای اصلی این سبک عبارتند از:

- **مخزن مرکزی داده ها** که در حقیقت یک ساختمان داده های بزرگ است که میان پردازش ها و بخشهای گوناگون به اشتراک گذاشته می شود.
- المان های پردازشی که به طور بالقوه مستقل از هم هستند. به این معنی که به کمک مخزن مرکزی داد ها قادرند همه نیازهای ارتباطی خود را برآورده ساخته و نیازی به ارتباط مستقیم با یکدیگر ندارند.

گونه بسیار ساده شده این روش در بسیاری شرکت ها استفاده می شود و آن به اشتراک گذاری فایل هاست. فایل های مورد نیاز اجزای مختلف (المان های پردازشگر مختلف) به همراه تعدادی ابزار اداری[۳۴] در اختیار این اجزا قرار می گیرد و مکانیزم معمولاً ساده ای برای تعیین دسترسی ها و جلوگیری از تداخل وضع می شود. در مورد سیستم های کوچک این روش ممکن است جواب مطلوب بدهد، اما با گسترش سیستم مسائلی نظیر همگام سازی[۳۵] داده ها درسرساز شده و جلوی مقیاس پذیری[۳۶] و گسترش این روش را می گیرند.

۲.۳.۲ سبک تخته سیاه (Blackboard)

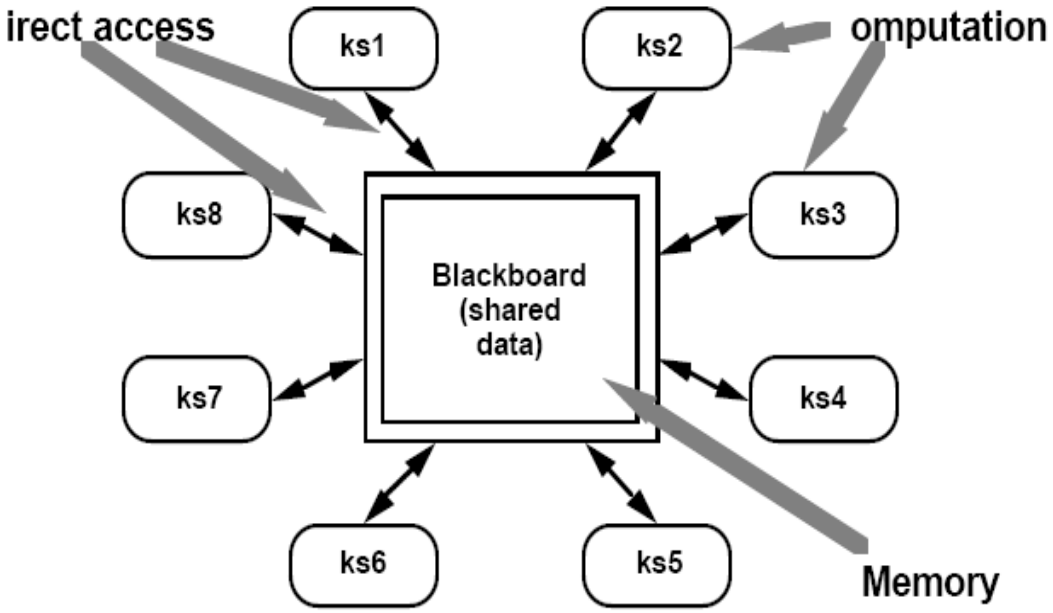
بطور کلی در سبک های داده-محور، سیاست های گوناگون می توانند انواع گوناگونی از سبک ها را تعریف کنند. به عنوان نمونه بر اساس شیوه دسترس خاصی که در روش Repository اتخاذ می شود، سبکی از معماری نرم افزار با نام تخته سیاه خلق می شود. در روش تخته سیاه، سه رکن اصلی حضور دارند:

- عناصر مستقل پردازش کننده اطلاعات که به آنها **منبع دانش**[۳۷] گفته می شود.
- **تخته سیاه** که در حقیقت همان ساختمان داده های مشترک است.
- **روش کنترل** که در واقع همان مدل فعال سازی است. در این زمینه در انتهای همین قسمت بیشتر صحبت خواهیم کرد.

منابع دانش به طور مستقیم با یکدیگر ارتباط برقرار نمی کنند و انتقال اطلاعات میان آنها از طریق تخته سیاه ممکن می –شود. معمولاً نحوه به کارگیری تخته سیاه توسط KS ها، کاملاً فرصت طلبانه است و به محض اینکه تخته سیاه به حالت آزاد رفت هر کدام از KS ها که بخواهند از روی آن خوانده، و یا بر روی آن اطلاعاتی را بنویسند، می کوشند تا

[33] Persistent Data
[34] Office Tools
[35] Synchronization
[36] Scalability
[37] Knowledge Source (KS)

تخته سیاه را به خدمت بگیرند، که از این میان تنها یکی موفق می شود و بقیه می بایست تا آزاد شدن مجدد تخته سیاه صبر کنند.



شکل ۵. سبک معماری تخته سیاه

یک نمونه از این سبک معماری در شکل ۵ نشان داده شده است. در این شکل، KS ها معرف منابع دانش مستقل از هم هستند که به تخته سیاه دسترسی مستقیم دارند.

و اما در مورد مدل فعال سازی، کلاً سه جزء از سیستم ممکن است بدانند کنترل اجرا به چه ترتیبی است:

- خود تخته سیاه: تخته سیاه نحوه انتقال کنترل را به عنوان یک داده کنترلی مشترک در اختیار همه منابع دانش گذاشته و آنها هم طبق توافق از این مدل کنترلی پیروی می کنند.
- منابع دانش: هر منبع می داند زمانی که با تخته سیاه کار دارد و همچنین تخته سیاه در وضعیت آزاد به سر می برد باید آنرا در اختیار بگیرد. این روش همان مدل رقابتی است که پیش تر توضیح دادیم.
- یک منبع خارجی به سیستم اضافه می کنیم که وظیفه آن اعمال مکانیزم کنترلی روی KS هاست.

۲.۳.۳ مثال هایی از بکارگیری سبک Repository و تخته سیاه

سبک تخته سیاه از قدیم عمدتاً برای کاربردهایی که نیاز به پردازش و تفسیر پیچیده سیگنال ها داشته اند نظیر پردازش صوت یا تشخیص الگو استفاده شده است. در مورد سبک Repository نیز در قدیم سیستم های دسته ای با بانک های اطلاعاتی اشتراکی وجود داشته اند که از این سبک معماری استفاده می کردند. همچنین کاربردهایی وجود دارند که در ابتدا تصور می شده سبک های دیگر معماری مناسب آنها هستند، امّا با گذشت زمان در عمل ثابت شده که سبک

معماری Repository به نحو بهتری می تواند چنین نرم افزارهایی را ساماندهی کند. به عنوان نمونه در ابتدا فرض می-شده که برای طراحی کامپایلرها بهترین سبک موجود، سبک *لوله ها و فیلترها* است، ولی امروزه عمده کامپایلرهای موجود در بازار بر اساس یک مخزن داده اشتراکی عمل می کنند (جدول نشانه ها[۳۸]، درخت ساختار نحوی، ...)

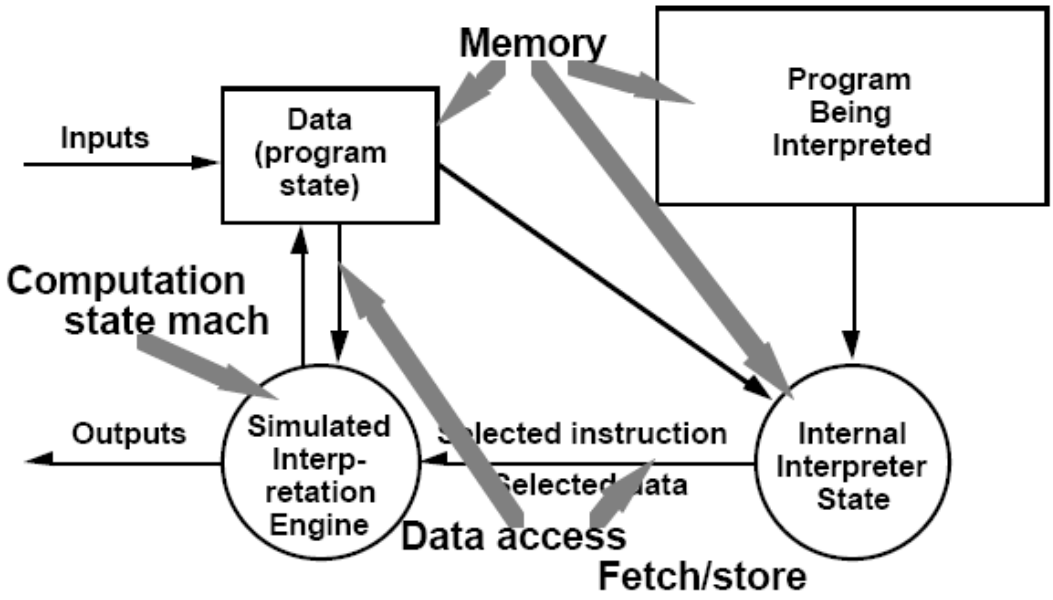
۲.۴ سبکهای ماشین مجازی (Virtual Machine)

سبک ماشینهای مجازی به دو شاخه *مفسرها* و *سیستم های* ***rule-based*** تقسیم می شود که ما فقط مفسرها را بررسی می کنیم:

۲.۴.۱ مفسرها (Interpreters)

مفسرها را گونه ای از ماشینهای مجازی می دانند. دلیل این امر آنست که یک مفسر می تواند یک لایه معنایی[۳۹] جدید روی تکنولوژی موجود درست کند، و این همان انتظاری است که از یک ماشین مجازی داریم. بخشهای مختلف یک مفسر عبارتند از:

- **موتور تفسیر**: وظیفه اصلی آن تفسیر و اجرای برنامه است.
- **حافظه**: که دربرگیرنده شبه برنامه[۴۰] می باشد.
- **نمایشی از وضعیت کنترلی موتور تفسیر**: این نمایش در بخشی از حافظه ذخیره سازی می شود.
- **نمایش وضعیت کنونی برنامه**: این نمایش هم در حافظه ذخیره سازی می شود.



شکل ۶. نمای کلی یک مفسر

[38] Symbol Table
[39] Semantic Layer
[40] Pseudo-Program

مدل فعال سازی در این سبک بستگی زیادی به نحوه پیاده سازی موتور تفسیر دارد. امّا به طور معمول موتورتفسیر دستورالعمل های یک شبه برنامه را ترتیبی به اجرا درمی آورد. همانند سبکهای دیگر معماری نرم افزار، این سبک را هم می توان به صورت ترکیبی در کنار چند سبک دیگر به کار برد. به عنوان یک مثال کلی، با ترکیب سبک مفسرها با سبک احضار ضمنی که از Trigger ها استفاده می کند، می توان یک موتور گردش کار ایجاد کرد که کنترل نحوه فعال شدن Trigger ها توسط مفسر انجام می شود و کاربر سیستم می تواند با ارائه شبه برنامه مورد نظر خود، رفتار سیستم را کنترل کرده و گردش کار مورد نیاز خود را پیاده سازی کند.

۲.۴.۲ مزایا و معایب مفسرها

- **مزایا**

- **انعطاف پذیری**: این سیستم ها می توانند بسیار انعطاف پذیر باشند، تا جایی که به عنوان مثال یک شرکت تولید نرم افزار می تواند برای مشتریانی که نیاز به مدل کردن گردش کار داشته و این گردش کار به تناوب دستخوش تغییر می شود، از یک نرم افزار مبتنی بر سبک مفسرها استفاده کند. به جای آنکه قوانین تجاری محیط هدف (محیطی که نرم افزار نهایتاً در آن نصب و استفاده می شود) به صورت استاتیک در نرم افزار گنجانده شود، کاربران سیستم می توانند با شبه برنامه هایی این قوانین را به دلخواه خود مدل کرده، تغییر دهند.

- **معایب**

- **پیچیدگی**: انعطاف پذیری بالای این گونه سیستم ها هزینه هایی را هم در هنگام تولید و تست به شرکت تولید کننده تحمیل می کند. بدیهی است که نمی توان تمام شبه برنامه هایی که کاربران ممکن است به عنوان ورودی به مفسر دهند را پیش بینی و امتحان کرد. در نتیجه هرگز نمی توان به طور کامل از درستی مفسر ساخته شده اطمینان حاصل کرد.

۲.۵ سبکهای مبتنی بر فراخوانی (Call/Return)

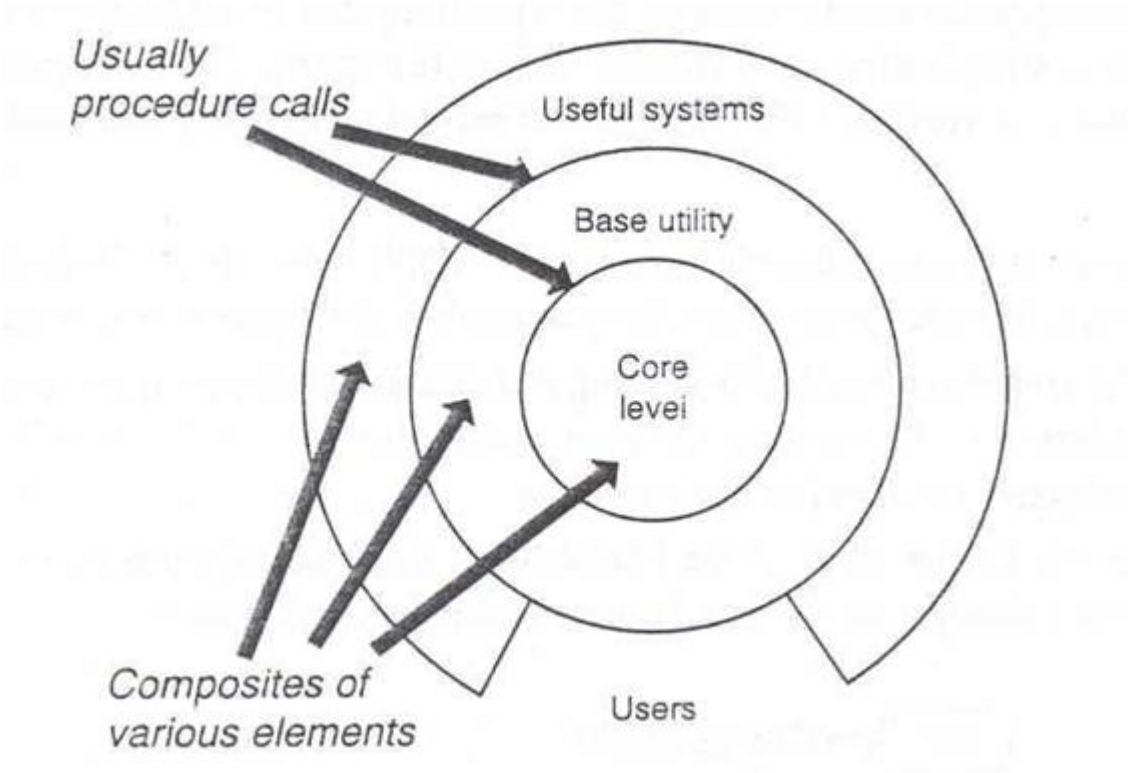
سه رده اصلی زیر مجموعه این خانواده از سبک ها عبارتند از: سبک لایه ای، سبک شئ گرا، و سبک Main Program and Sub-routine (سبک کلاسیک زبان های برنامه نویسی) . مشخصه مشترک و اساسی همه این سبکها، مدل فعال-سازی یکسان آنهاست. در همه این سبکها یک رشته اصلی[41] کنترل وجود دارد که این رشته فراخوانی عملیات و متدهای دیگر را بر عهده دارد.

۲.۵.۱ سبک معماری لایه ای

سیستم های لایه ای ذاتاً ماهیت سلسله مراتبی دارند. به این معنی که یک لایه خدماتی را برای لایه بالایی خود فراهم کرده و به عنوان مشتری از لایه پایینی خود سرویس دریافت می کند. لایه ها ممکن است از نظر میزان شفافیت

[41] Main Thread

(دسترس پذیری) با یکدیگر متفاوت باشند. در بعضی از سیستم ها هر لایه از دید تمام لایه ها بجز لایه خارجی تر مجاور پنهان است، البته گاهی به دلایل فنی برخی از توابع خاص با دقت انتخاب شده و در دسترس سایر لایه ها هم قرار می-گیرند. به این صورت، اجزای نرم افزار یک ماشین مجازی را تشکیل می دهند (همانگونه که در بحث مفسرها هم ذکر شد).



شکل ۷. یک سیستم لایه ای - لایه های مختلف میزان شفافیت های مختلفی را هم برای کاربر فراهم می کنند.

نحوه ارتباطات میان لایه ها در این سبک، توسط پروتکلهای ارتباطی تعیین می شود. مشهورترین مثال های این سبک معماری، پروتکل های لایه ای ارتباطی نظیر TCP/IP و یا مدل مرجع ISO هستند. البته این سبک معماری کاربردهای دیگری نظیر طراحی سیستم های عامل و یا سیستم های مدیریت پایگاه داده ها هم دارد. همچنین یک مدل معروف و ۳ لایه ای از این سبک که به مدل 3-Tier شهرت یافته، در طراحی نرم افزارهای تجاری فراوان مورد استفاده قرار می-گیرد.

- **مزایا**
  - طراحی نرم افزار در این سبک بر اساس افزایش تدریجی سطح انتزاع انجام می شود، و این روند به دلیل سازگاری با مدل طبیعی ادراک بشر می تواند فرآیند طراحی را ساده کند.
  - به دلیل Modularity بالا بهبود نرم افزار، در این روش به سادگی انجام می شود (مشابه سبک لوله –ها و فیلترها).
  - استفاده مجدد از نرم افزار در این سبک پشتیبانی می شود.

- **معایب**

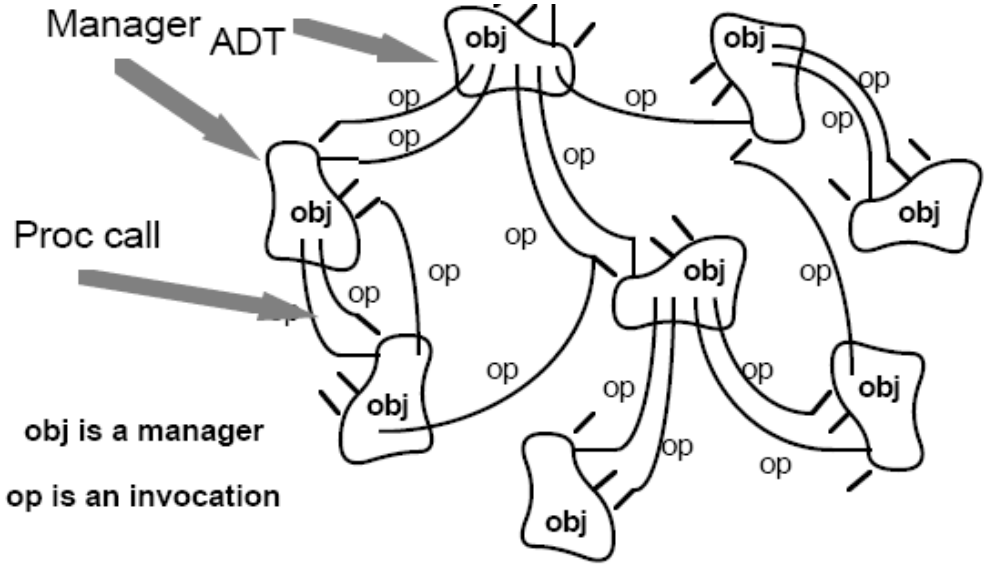
o تمام سیستم های نرم افزاری را نمی توان به سهولت به لایه هایی تقسیم کرد و این امر می تواند کاربرد این سبک را تا محدود به موارد خاص کند.

o کارایی ممکن است درسرساز شود. سؤال اصلی اینجاست که چه تعداد لایه داشته باشیم که همچنان سیستم در عمل قابل پیاده سازی باشد و منطق سطح بالای سیستم از پیاده سازی سطح پایین آن آنقدر فاصله نگیرد که در هنگام اجرا کارایی بسیار ضعیفی حاصل شود. این همان مشکلی است که مدل مرجع ISO تا حد زیادی به آن گرفتار شد.

o منتقدین این سبک معتقدند لایه بندی یک سبک نیست و تنها مشخصه ای طبیعی از سیستم های با ساختار سلسله مراتبی است [۲].

۲.۵.۲ سبک معماری شئ گرا (Object Oriented)

شئ گرایی عمدتاً با مفهوم Encapsulation شناخته می شود. منظور از Encapsulation تجمع عملیات اولیه در کنار نوع داده های انتزاعی[۴۲] در قالب موجودیتی به نام شئ است. برخی نویسندگان [۳] اشیاء را اجزایی از نرم افزار می –دانند که نقش مدیریتی دارند. به این معنی که مسئولیت حفظ جامعیت کل سیستم بر عهده آنهاست و ویژگی هایی نظیر پنهان سازی اطلاعات توسط آنها فراهم می شود. یک شئ اصولاً به اشیاء دیگر اجازه دسترسی به داده های خصوصی[۴۳] خود را نمی دهد (البته بجز به اشیائی از کلاسهای دوست) مگر آنکه پیغام درخواستی از طرف آنها دریافت کند و در ضمن، عملیاتی برای فراهم کردن آن داده بخصوص در پاسخ به پیغام درخواست، در شئ دریافت کننده پیغام تعبیه شده باشد. به این ترتیب پنهان سازی اطلاعات توسط یک شئ تضمین می شود.



شکل ۸. سبک شئ گرا – ارتباط اشیا تنها از طریق عملیات (در واقع از طریق درخواست عملیات به کمک پیغام ها) امکان پذیر است.

[42] Abstract Data Type (ADT)
[43] Private Data

- **مزایا**
  - به بهترین شکل ممکن استفاده مجدد از نرم افزار را پشتیبانی می کنند.
  - پنهان سازی اطلاعات سبب می شود ارتباط میان اجزای نرم افزار ساده تر شود.
  - از مدل فرآیند گرای[۴۴] کلاسیک که کل سیستم را در قالب تعدادی تابع پیاده سازی می کرد فاصله گرفته و به داده ها اهمیت داده شده است.
  - دنیای واقعی با این سبک بهتر مدل می شود.
- **معایب**
  - بر خلاف مدل هایی نظیر لوله ها و فیلترها، در این سبک اشیاء برای آنکه بتوانند با یکدیگر تعامل داشته باشند، می باید از وجود هم و همچنین واسط هایی که ارائه می کنند، باخبر باشند. به این ترتیب اشیاء به یکدیگر وابستگی پیدا می کنند.

۲.۵.۳ سبک معماری Main Program and Sub-routine

در این سبک معماری، ساماندهی سیستم تولید شده تا حد زیادی ماهیت زبان برنامه نویسی که در پیاده سازی سیستم به کار رفته است را منعکس می کند. البته استفاده از این سبک در قدیم که برنامه ها چندان پیچیده نبودند متداول بوده و امروزه، حداقل در مورد نرم افزارهای تجاری بزرگ، به عنوان یک سبک معماری نرم افزار به کار نمی رود.
مشخصه اصلی این سیستم ها آنست که یک برنامه اصلی گرداننده کل سیستم بوده و در مواقع ضروری زیربرنامه هایی را فراخوانی می کند. عمدتاً این سیستم ها از نبود Modularization مطلوب رنج می برند و در نتیجه قابلیت توسعه چندانی ندارند. به دلیل محدودیت هایی که سیستم های تولید شده بر اساس این سبک دارند، بیش از این به بحث در مورد آن نمی پردازیم.

۳ سایر سبکهای معماری نرم افزار

۳.۱ سبک فرآیندهای توزیع شده (Distributed Processes)

سیستم های توزیع شده، سازماندهی های ویژه ای را برای سیستم های چندپردازنده ای[۴۵] بوجود آورده اند. بعضی از این سازماندهی ها با توپولوژی خاصی که دارند توصیف می شوند، نظیر توپولوژی حلقه[۴۶]، و یا توپولوژی ستاره[۴۷]. برخی دیگر توسط پروتکل های خاصی که برای ارتباط میان پردازه ها به کار می روند، شناخته شوند. گاهی سیستم های Client/Server را به عنوان نمونه خاصی از این سبک معرفی می کنند. در چنین سیستم هایی سرور معمولاً پردازشی است که سرویسی را برای پردازش های دیگر فراهم می کند. با این حال عمدتاً این Client است که از وجود سرور و مشخصات آن آگاه است (یا از طریق سرور دیگری می تواند این اطلاعات را کسب کند)، و نه سرور.

۳.۲ سبکهای خاص منظوره (Domain-Specific)

[44] Process-oriented
[45] Multi-processor Systems
[46] Ring Topology
[47] Star Topology

در دهه اخیر تلاشهای فراوانی به عمل آمده است که برای زمینه های خاص، مدل مرجع معماری تهیه شود. چنین مدلهایی، ساختارهای مخصوصی برای یک خانواده از کاربردها ارائه می کنند، به عنوان مثال برای سیستم های حمل نقل یک مدل معماری ویژه پیشنهاد می نمایند. با تخصصی تر شدن مدل معماری، این امکان به وجود می آید که یک سیستم قابل اجرا و استفاده، به واسطه محدودیت های خوش تعریف اعمال شده روی توصیف آن به طور کاملاً خودکار و یا نیمه خودکار تولید شود.

۳.۳ سبک انتقال حالت (State Transition)

سبک معمولی برای سیستم های واکنشی است. اجزای اصلی این سبک یک مجموعه از حالت ها به همراه مجموعه ای از انتقال حالت ها است. منظور از انتقال حالت، تغییر وضعیت سیستم از حالتی به حالت دیگر است (مشابه آنچه در State Transition Diagrams (STD) در روشهای ساخت یافته تولید نرم افزار و یا در Activity Diagram های UML دیده می شود).

۳.۴ سبک کنترل فرآیند (Process Control)

این سبک بیشتر در سیستم هایی که وظیفه آنها کنترل یک محیط فیزیکی (نظیر کنترل شرایط رطوبت و دما و ... در یک گلخانه) است مورد استفاده قرار می گیرد. اگر بخواهیم بسیار کلی این چنین سیستم هایی را توصیف کنیم، اجزای تشکیل دهنده آنها عبارت خواهند بود از:

- **سنسورها**: اطلاعات بازخور[۴۸] را از محیط دریافت کرده و به واحدهای پردازشی می رساند.
- **واحد(های) کنترل فرآیند**: متناوباً اطلاعات دریافتی از سنسورها را مورد پردازش قرار داده و در صورت مشاهده انحراف در وضعیت کل سیستم به نسبت وضعیت مطلوب ازیش تعیین شده، تصمیمات لازم را اتخاذ می کند.
- **عملگرها**[۴۹]: تصمیمات اتخاذ شده توسط واحد یا واحدهای کنترل فرآیند را در محیط به اجرا در می آورند.

۴ منابع

[1]. Len Bass, Paul Clementsm Rick Kazman, "Software Architecture in Practice", 2nd Edition, 2003.

[2]. Stephen T. Albin, "The Art of Software Architecture: Design Methods and Techniques", 2003.

[3]. Mary Shaw and David Garlan, "An Introduction to Software Architecture", 1994.

[4]. http://www.wikipedia.org

[5]. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", 1994.

[48] Feedback
[49] Actuators