به نام خدا

درس الگوها در مهندسی نرم افزار

تمرین دوم

استاد: جناب آقای دکتر رامسین

عطیه محمدخانی

بهار ۱۴۰۳

فهرست

۱– شرح مساله اول

ابتدا به طرح کلی مساله میپردازیم، سپس با دقت به بررسی تک تک اجزای صورت سوال پرداخته و راهکارهای قابل تامل را بررسی و شرح می‌دهیم.

"*در یک سیستم تطبیق پذیر توزیع شده، تعدادی از زیرسیستم های Utility با یکدیگر برای ارائه سرویس‌ها به زیرسیستم های مشتری همکاری می کنند. هدف این است که سیستم را طوری طراحی کنیم که سرویس‌ها ارائه شده به هر مشتری بر اساس وضعیت کلی سیستم پیکربندی شود و در زمان اجرا قابل پیکربندی مجدد باشد. به منظور پیاده سازی هر سرویس، زیرسیستم های Utility درگیر باید در یک زنجیره به هم مرتبط شوند.*"

نیازمندی های اصلی(Force):

۱. سیستم باید توزیع شده باشد. *(الگوی پیشنهادی Microkernel)*
۲. سیستم باید تطبیق پذیرAdapter باشد.(*الگوی پیشنهادی Adapter)*
۳. همکاری در زیرسیستم های Utility وجود دارد.(*الگوی پیشنهادی Façade)*
۴. سرویس‌ها ارائه شده به هر مشتری بر اساس وضعیت کلی سیستم پیکربندی شود.(ا*لگوی پیشنهادی State)*
۵. مانیتور کردن زیر سیستم ها و جمع آوری اطلاعات از هر زیر سیستم. (*الگوی پیشنهادی Observer)*
۶. قابل پیکربندی مجدد در زمان **اجرا** باشد.(*الگوی پیشنهادی strategy)*
۷. ارتباط زیرسیستم های Utility درگیر یک زنجیره را تشکیل می دهد.*(الگوی Chain of responsibility)*

۱–۱ توزیع شدگی

در صورت مساله بیان شده که این سیستم، سیستمی تطبیق پذیر و توزیع شده است. طبق آنچه از درس آموخته ایم بلافاصله سه الگو برای مدیریت توزیع شدگی به ذهن میرسد. الگوهای سیستم های توزیع شده زیرساخت لازم برای اپلیکیشن های توزیع شده را در اختیار می گذارد. شامل الگوهای Broker و Microkernel و Pipe&Filter است.

الگوی معماری Pipe&Filter که با استفاده از ایجاد یک خط لوله توزیع شده و موازی سازی پردازش سعی در سریع حل کردن یک مساله ی توزیع شده همگن دارد. سیستمی است که قرار است جویبار های داده را پردازش کند به هر گام پردازشی اصطلاحا فیلتر می گویند و بین خود فیلترها به صورت خط لوله یک زنجیره قرار می گیرد و داده ها با المان‌هایی به نام پایپ منتقل می شوند و بین هر دو فیلتر مجاور یک پایپ وجود دارد. بنابراین اصل پردازش توسط فیلترها انجام و پایپ ها صرفا انتقال دهنده داده هستند. البته پایپ ها رفتارهایی مثل synchronization هم انجام دارند که زمانی که فیلترها کاملا موازی اجرا می شوند. هدف اصلی الگو را که پیکربندی و بازپیکربندی است را فراهم می کنند و باید بتوان ساختار را به سادگی در زمان تغییر داد و پیکربندی مجدد نمود. وقتی از این الگو استفاده می کنیم که بتوان processing incremental کرد یعنی هر گام داده را گرفت و پردازشی کرد و داده را به بعدی فرستاد و مجبور نباشیم برای تولید خروجی بخش بزرگ ورودی ها را بگیریم. مهم ترین هدف الگو این است که می خواهیم کمترین coupling را بین فیلتر ها داشته باشیم تا بتوانی ترکیب گام های پردازشی را هر طور که خواستیم بچینیم.

الگوی معماری Broker که با استفاده از proxy های سمت کلاینت و سرور، broker service و همینطور bridge مساله توزیع شدگی را به گونه ای حل میکند که اجزای مختلف از مکان یکدیگر خبر نداشته باشند و سرویس‌ها با interfaceهای مختلف بتوانند با یکدیگر به تعامل بپردازند. Broker و Bridge دو عنصر اصلی در این الگو هستند. brokerها پیام رسان هستند و درخواست را ازکلاینت گرفته و به سرور می رسانند و پاسخ یا خطاها را به کلاینت برمی‌گردانند و این کار را در محیط توزیع شده بدون اینکه کلاینت و سرور از پیچیدگی های محیط توزیع شده باخبر شوند انجام می‌دهند bridge ها اجباری نیستند و جایی استفاده می‌شوند که broker ها می خواهند با هم ارتباط برقرار کنند اما زبان هم را نمی فهمند و سیستم وپروتکل آنها متفاوت است bridgeها دغدغه ارتباط بخش های نامتجانس را بر عهده می گیرند تا broker ها مجبور به این کار نباشند و بایدبدانند چطور ارتباط برقرارکنند. همینطور

ارتباطات بین سرویس‌های روی چند گره شبکه را از طریق bridge به گونه ای به انجام برسانیم که هیچ وابستگی مستقیمی بین آنها برقرار نشود عناصر دیگر موجود در الگوکلاینت هاو سرور ها هستند. سرور ها اشیایی هستند که سرویس می دهند و وظیفه مندی را در اختیار دیگران از طریق اینترفیس میگذارند و کلاینت و سرور در دوفضای مختلف هستند. در اینجا تعدادی زیر سیستم داریم که با یک زیرسیستم مشترک در حال تعامل هستند. هر زیرسیستم اینجا یک کلاینت است و زیرسیستم مشترک سرور است. همچنین در سمت کلاینت Proxy Side-Client داریم که نقش سرور را برای کلاینت بازی می‌کند. در سمت سرور هم یک Proxy Side-Server داریم که نقش کلاینت را برای سرور دارد.

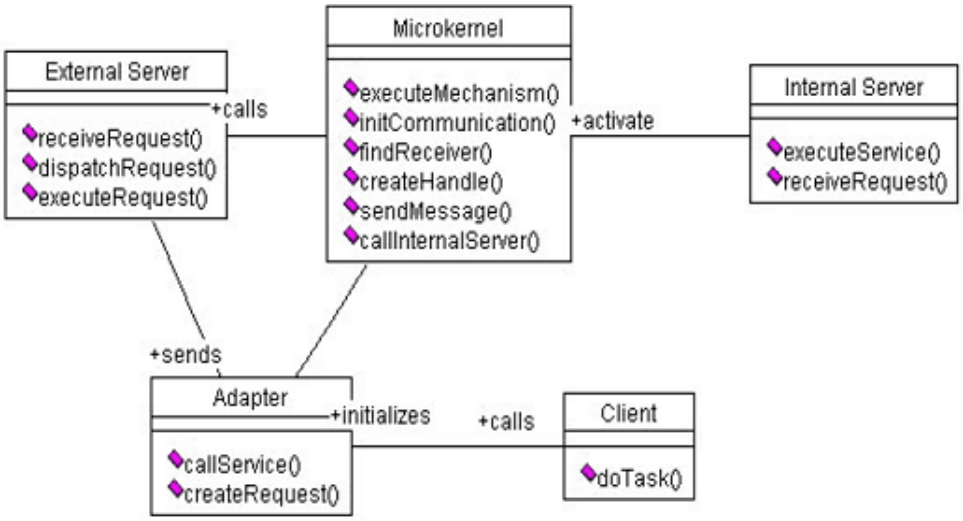
الگوی معماری Microkernel برای سیستمهای توزیع شده تطبیق پذیر می باشد به طوری که یک هسته وظیفه‌مندی کمینه با سرویس های حداقلی در نظرگرفته و آنرا درmicrokernel قرار می دهیم و با ترکیب آنها سرویس های درشت دانه ساخت و آن را در اختیار کلاینت بیرونی قرار داد. کلاینت می تواند کلاس بیرونی یا اپلیکیشن بیرونی باشد. مخصوصا برای سیستمهایی که باید adaptable باشند یعنی نیازمندی ها تغییر می کنند الگو مناسب است. ولی در محیط توزیع شده microkernel جایی است که مرکزیت است که منابع پردازشی و ذخیره سازی را در اختیار می گذارد و تمام ارتباطات هم در اختیارش است. Microkernel که با استفاده از ایجاد یک مجموعه کوچک از functionalityهای پایه ای و ارائه باقی نیازمندی ها با استفاده از این مجموعه پایه ای سعی در حل مساله توزیع شدگی برای سیستم های با نیازمندی های دائما در حال تغییر دارد.

به منظور ایجاد توزیع شدگی الگوی Microkernel را پیشنهاد می کنیم.

۱-۲ تطبیق پذیری

از آنجایی که الگوی معماری میکروکرنل به عنوان الگوی معماری پلاگین نیز شناخته می شود برای سیستم‌های نرم‌افزاری که باید قادر به تطبیق با نیازهای متغیر سیستم باشند، کاربرد دارد. معمولاً زمانی استفاده می شود که تیم های نرم افزاری نیاز دارند تا سیستم هایی با اجزای قابل تعویض ایجاد می کنند. این یک هسته عملکردی حداقلی را از عملکرد گسترده و بخش های خاص مشتری جدا می کند. همچنین به عنوان سوکتی برای اتصال این افزونه ها و هماهنگی همکاری آنها عمل می کند. الگوی معماری میکروکرنل این امکان را می دهد که ویژگی های برنامه اضافی را به عنوان افزونه به برنامه اصلی اضافه و قابلیت توسعه و جداسازی و جداسازی ویژگی ها را فراهم می کند.

الگوی معماری میکروکرنل از دو نوع جزء معماری تشکیل شده است: یک سیستم اصلی و ماژول های پلاگین. منطق برنامه بین ماژول‌های پلاگین مستقل و سیستم اصلی اصلی تقسیم می‌شود و قابلیت توسعه، انعطاف‌پذیری و جداسازی ویژگی‌های برنامه و منطق پردازش سفارشی را فراهم می‌کند. و سیستم اصلی الگوی معماری میکروکرنل به طور سنتی فقط حداقل عملکرد مورد نیاز برای عملیاتی کردن سیستم را شامل می شود. نمای کلی دیاگرام کلاس میکروکرنل در شکل ۱ آماده است.



شکل ۱- نمای کلی دیاگرام کلاس الگوی میکروکرنل

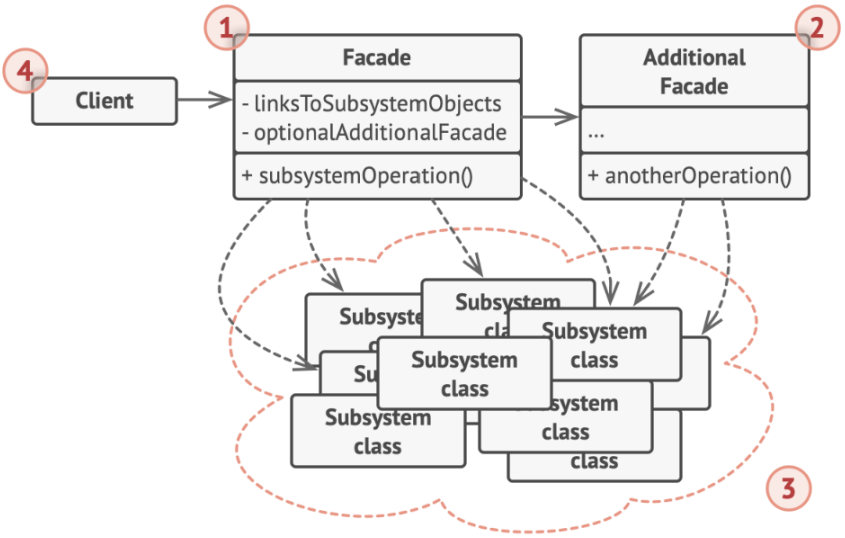
مزایای:

- ✓ انعطاف پذیری و توسعه پذیری عالی
- ✓ امکان اضافه کردن پلاگین‌ها
- ✓ قابلیت حمل خوب
- ✓ سهولت در استقرار
- ✓ پاسخ سریع به یک محیط دائما در حال تغییر
- ✓ کارایی بالا(می توان برنامه ها را سفارشی و ساده کنیم تا فقط ویژگی هایی را که نیاز دارید را شامل شود.)

۱-۳ همکاری داخل زیرسیستم ها

به منظور ایجاد همکاری در بین اکسترنال سرورها الگوی Façade پیشنهاد می گردد:

Façade یا نما یک الگوی طراحی ساختاری است که یک رابط ساده برای یک کتابخانه، یک چارچوب یا هر مجموعه پیچیده دیگری از کلاس ها ارائه می دهد. نما دسترسی راحت به بخش خاصی از عملکرد زیرسیستم را فراهم می‌کند. می داند که درخواست مشتری را به کجا هدایت کند و چگونه تمام قطعات متحرک را کار کند. زیرسیستم پیچیده از ده‌ها شی مختلف تشکیل شده است. برای اینکه همه آنها کاری معنادار انجام دهند، باید عمیقاً در جزئیات پیاده سازی زیرسیستم را بداند، مانند مقداردهی اولیه اشیاء به ترتیب صحیح و ارائه داده ها به آنها با فرمت مناسب. طبقات زیرسیستم از وجود نما آگاه نیستند. آنها در داخل سیستم عمل می کنند و مستقیماً با یکدیگر کار می کنند. Client به جای فراخوانی مستقیم اشیاء زیرسیستم از نما استفاده می‌کند.



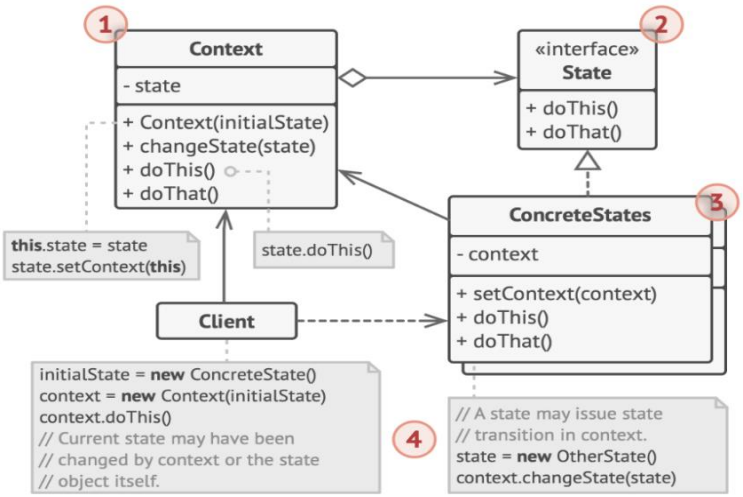
شکل۲- نمای کلی کلاس دیاگرام الگوی فَساد

۱ -۴ ارائه سرویس‌ها به هر مشتری بر اساس وضعیت کلی سیستم

به منظور ارائه سرویس‌ها به هر مشتری بر اساس وضعیت کلی سیستم الگوی State، انتخاب پیکربندی مناسب الگوی Strategy ومشاهده وضعیت سیستم الگوی Observer پیشنهاد می گردد:

در سیستم تعدادی State مختلف داریم که براساس آنها می توان ارائه سرویس‌ها به هر مشتری بر اساس وضعیت سیستم تامین کرد. هر کدام از این State ها بر اساس Observer یک سری مانیتور در سیستم هستند که اطلاعات مختلف را در سیستم جمع آوری می کنند. مانیتور ها پس از تغییر حالت، این تغییر حالت ها به State مورد نظرش که observer آن است ارسال و State با دریافت حالت مانیتور می تواند state را تغییر بدهد. تغییر این state می تواند منجر به تغییر مکانیزم سرویس‌ها به مشتری بشود یا ممکن است شرایط سیستم تغییری نکند. در صورت تغییر به سیستم مرکزی اطلاع داده می شود تا استراتژی را تغییر و پیکربندی مجدد انجام دهد.

الگوی State پیشنهاد می‌کند که کلاس‌های جدیدی برای همه حالت‌های ممکن سیستم ایجاد و همه رفتارهای خاص حالت را در این کلاس‌ها استخراج می‌کنیم. به جای اجرای همه رفتارها به شی اصلی، به نام context، ارجاعی به یکی از اشیاء حالت سیستم که نشان دهنده وضعیت فعلی آن است را ذخیره می کند و تمام کارهای مربوط به حالت را به آن شی واگذار می کند. Context از طریق رابط حالت با شیء حالت ارتباط برقرار می کند. Context یک تنظیم کننده را برای ارسال یک شی حالت جدید به آن نشان می دهد. قوانین سوئیچینگ که انتقال نامیده می شوند نیز متناهی و از پیش تعیین شده هستند.

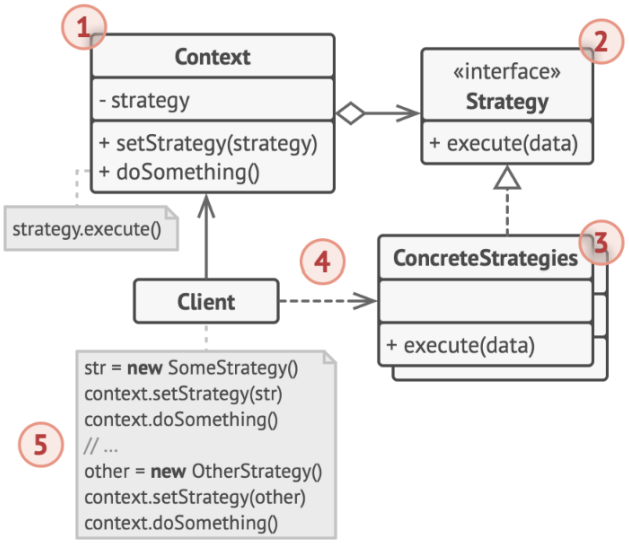


شکل ۳- نمای کلی کلاس دیاگرام الگوی وضعیت

۱. Context یک ارجاع به یکی از اشیاء حالت مشخص را ذخیره می کند و تمام کارهای مربوط به حالت را به آن واگذار می کند. زمینه از طریق رابط حالت با شیء حالت ارتباط برقرار می کند.
۲. رابط حالت روش های خاص حالت را اعلام می کند.
۳. زیرکلاس های وضعیت پیاده سازی های خود را برای روش های خاص وضعیت ارائه می کنند.
۴. اشیاء حالت ممکن است یک مرجع بازگشتی به شی زمینه ذخیره کنند. از طریق این مرجع، حالت می تواند هر گونه اطلاعات مورد نیاز را از شی زمینه واکشی کند، و همچنین انتقال حالت را آغاز کند.
۵. هر دو حالت متن و انضمام می توانند حالت بعدی زمینه را تنظیم کنند و با جایگزین کردن شی حالت مرتبط با زمینه، انتقال حالت واقعی را انجام دهند.

الگوی استراتژی

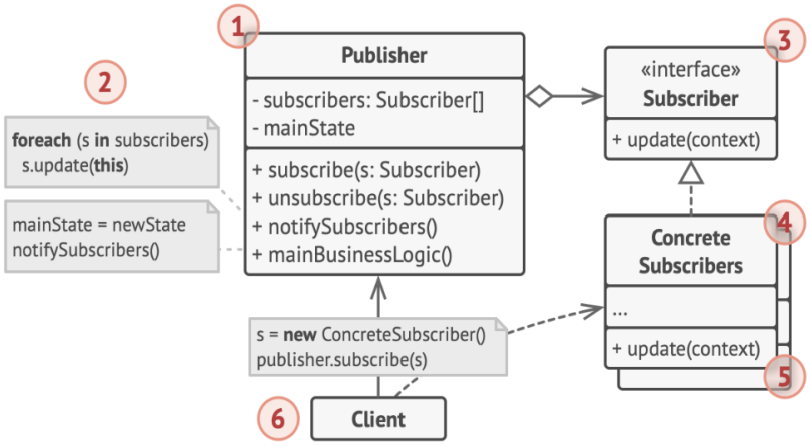
۱. Context یک ارجاع به یکی از استراتژی های مشخص را حفظ می کند و تنها از طریق رابط استراتژی با این شی ارتباط برقرار می کند.
۲. رابط استراتژی برای همه استراتژی های مشخص مشترک است. روشی را که زمینه برای اجرای یک استراتژی استفاده می کند، اعلام می کند.
۳. زیرکلاس‌های استراتژی تغییرات مختلفی از الگوریتمی را که زمینه استفاده می کند، پیاده سازی می کند.
۴. زمینه، هر بار که نیاز به اجرای الگوریتم دارد، متد اجرا را روی شی استراتژی پیوند یافته فراخوانی می کند. زمینه نمی داند با چه نوع استراتژی کار می کند یا الگوریتم چگونه اجرا می شود.
۵. Client یک شی استراتژی خاص ایجاد می کند و آن را به متن ارسال می کند. زمینه یک تنظیم کننده را نشان می دهد که به مشتریان اجازه می دهد استراتژی مرتبط با زمینه را در زمان اجرا جایگزین کنند.



شکل ۴- نمای کلی کلاس دیاگرام الگوی استراتژی

الگوی Observer

الگوی استراتژی با هدف تعریف خانواده ای از الگوریتم ها یا پیکربندی ها، هر کدام را کپسوله و آنها را قابل تعویض می‌کند.
- استفاده راهبردهای مختلف را برای پیکربندی سرویس ها بر اساس وضعیت فعلی سیستم یا الزامات خاص را مجاز می کند. استراتژی های پیکربندی مختلف را می توان در زمان اجرا بدون تغییر کد تعریف و تعویض کرد. مشتریان می توانند در تغییرات وضعیت مشترک شوند و هنگامی که از تغییرات مربوطه مطلع شدند، سرویس‌ها خود را در زمان واقعی پیکربندی مجدد کنند.



شکل ۵- نمای کلی کلاس دیاگرام الگوی مشاهده گر

۱-۵ قابلیت پیکربندی مجدد در زمان اجرا

به منظور قابلیت پیکربندی مجدد در زمان اجرا الگوی استراتژی، وضعیت و آداپتور پیشنهاد می گردد:

با کمک آداپتور موجود در الگوی میکروکرنل می توان سرویس‌ها را به صورت پویا بر اساس وضعیت به سیستم اضافه یا کم کرد. در مورد پیکربندی مجدد در زمان اجرا با استفاده الگوی وضعیت و استراتژی در بخش قبلی بحث شد.

۱-۶ تشکیل زنجیره بین زیرسیستم های درگیر و همکار

به منظور تشکیل زنجیره بین زیرسیستم‌های درگیر در زمان اجرا الگوی Chain of Responsibility پیشنهاد می گردد:

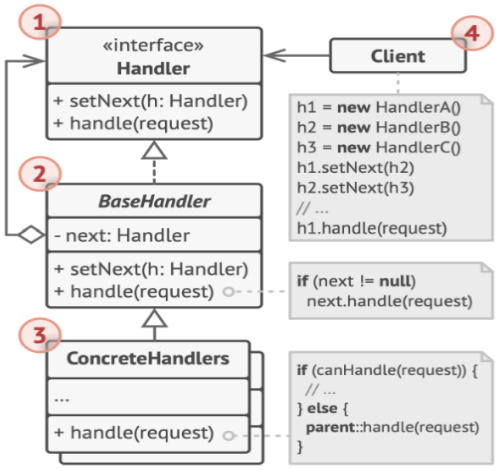
به منظور ایجاد زنجیره ای از زیرسیستم های کاربردی که درخواست ها را به ترتیب پردازش می کند از الگوی زنجیره مسئولیت استفاده می‌شود. این الگو پیوند زیرسیستم های کاربردی را پیاده سازی می کند که در آن هر زیرسیستم می تواند درخواستی را انجام دهد یا آن را به زیرسیستم بعدی در زنجیره ارسال کند. هر زیرسیستم utility می تواند یک کنترل کننده در زنجیره باشد، بخشی از یک درخواست سرویس را پردازش کرده و در صورت نیاز آن را ارسال کند.

Handler، اینترفیسی را برای همه کنترل کننده های زیرکلاس های هندلر مشترک اعلام می کند.

Base Handler یک کلاس اختیاری است که برای همه زیرکلاس های هندلر مشترک است.

Concrete Handlers حاوی کد واقعی برای پردازش درخواست ها است. به محض دریافت درخواست، هر کنترل کننده باید تصمیم بگیرد که آیا آن را پردازش کند و علاوه بر این، آیا آن را در طول زنجیره ارسال کند یا خیر.

بسته به منطق برنامه، Client ممکن است فقط یک بار زنجیره بسازد یا آنها را به صورت پویا بسازد. یک درخواست می تواند به هر کنترل کننده در زنجیره ارسال شود - لازم نیست اولین درخواست باشد.



شکل ۶- نمای کلی کلاس دیاگرام الگوی زنجیره مسئولیت

۱ –۷ پایین بودن وابستگی

استفاده از الگوهای طراحی میکروکرنل و ساختارهای کلاس آداپتور، خارج شدن سرویس های اضافی در اینترنال سرور و سرورهی تخصصی در اکسترنال سرور همگی باعث ایجاد coupling indirect شده و وابستگی مستقیم بین عناصر محیط را از بین میبرد. صرفا عناصری که وابستگی ذاتی به یکدیگر دارند مثل میکروکرنل و سرورهای اینترنال و اکسترنال و با یکدیگر منسجم میمانند که ایرادی ندارد و باعث ایجاد تغییر منتشر شونده نخواهد شد. البته می توان برای آنان سرورساید پراکسی نیز در نظر گرفت. این پایین بودن وابستگی و بالا بودن انسجام ناشی از اعمال الگوهایی مانند آداپتور، سبب میشود که این سیستم، مصداق برآورده شدن اصل OCP در شی گرایی باشد.

۱ –۸ توسعه پذیری آسان سیستم

همانطور که در بخش قبل بیان شد. این سیستم میتواند مصداقی از برآورده شدن اصل OCP در شی گرایی باشد. در واقع یعنی گسترش سیستم باعث ایجاد نیاز به تغییرات نامرتبط و به خصوص تغییرات منتشر شونده نخواهد شد و باعث توسعه پذیری آسان سیستم خواهد شد.

۱ –۹ نامرئی بودن موقعیت مکانی

یک میکروکرنل با انتزاع و جداسازی ارتباط بین سرویس ها و فرآیندها، بدون توجه به موقعیت فیزیکی آنها در سیستم، شفافیت مکان را فراهم می کند. این بدان معناست که سرویس‌ها می‌توانند به گونه‌ای تعامل داشته باشند که گویی روی یک دستگاه هستند، حتی اگر در میان سخت‌افزارها یا گره‌های شبکه مختلف توزیع شده باشند. میکروکرنل جزئیات ارسال پیام را کنترل می کند و موقعیت سرویس ها را برای کاربر و برنامه شفاف می کند. این انتزاع طراحی سیستم های توزیع شده را ساده می کند و انعطاف پذیری و مقیاس پذیری را افزایش می دهد

۱ –۱۰ مدیریت پیام ها

میکروکرنل ها شفافیت مکان را از طریق مفهومی به نام ارسال پیام ارائه می کنند. در معماری میکروکرنل، سیستم به اجزای کوچک و مدولار تقسیم می شود که تنها ضروری ترین توابع در حالت هسته اجرا می شوند. این طراحی هسته را به حداقل می رساند و اغلب به وظایفی مانند ارتباطات بین فرآیندی (IPC) و مدیریت حافظه محدود می شود.

هنگامی که یک فرآیند در یک سیستم مبتنی بر میکروکرنل نیاز به ارتباط با فرآیند دیگر یا دسترسی به یک منبع دارد، این کار را از طریق ارسال پیام انجام می دهد. پیام ها بین فرآیندها یا بین یک فرآیند و یک سرویس ارائه شده توسط میکروکرنل ارسال می شوند. این ارتباط بدون توجه به موقعیت فیزیکی فرآیندها یا منابع درگیر اتفاق می افتد.

از آنجایی که فرآیندها از طریق واسط ها و پیام های کاملاً تعریف شده با یکدیگر ارتباط برقرار می کنند، مستقیماً به مکان فیزیکی سایر فرآیندها یا منابع وابسته نیستند. این لایه انتزاعی ارائه شده توسط میکروکرنل شفافیت مکان را امکان پذیر می کند، به این معنی که فرآیندها می توانند بدون نیاز به دانستن اینکه در کجا به طور فیزیکی در سیستم اجرا می شوند با یکدیگر تعامل داشته باشند. این باعث انعطاف پذیری، مقیاس پذیری و مدیریت آن آسان تر می شود.

۲– معماری پیشنهادی

در این بخش ابتدا فرضیاتی که در حل مساله در نظر گرفتهایم را شرح داده و سپس به شرح معماری پیشنهادی و اجزای آن میپردازیم. تلاش میکنیم با استفاده از نمودارهای کلاس و توالی رخداد ، نسبت به ساختار و رفتار معماری پیشنهادی در سناریوهای مختلف شهود کافی را منتقل کنیم. در نهایت معماریای که ارائه کردهایم را مورد نقد قرار داده و تلاش میکنیم از معایب آن سخن بگوییم.

۲–۱ توصیف معماری

الگوی میکروکرنل یک هسته عملکردی حداقلی را از قابلیت هایی که توسعه می یابند و بخش های مربوط به مشتری جدا می کند. پشتیبانی از طراحی سیستم‌عامل‌های کوچک، کارآمد و قابل حمل، و پشتیبانی از گسترش سرویس‌ها جدید، به بسیاری از سیستم‌ها کمک می‌کند که به درجه بالایی از سازگاری با پلتفرم‌های مختلف و نیازهای خاص مشتری نیاز دارند. برای سیستم های نرم افزاری که باید قادر به تطبیق با نیازمندی های سیستم در حال تغییر باشند، کاربرد دارد.

این الگو در شرایطی استفاده می شود که توسعه چندین برنامه کاربردی که از رابط های برنامه نویسی مشابهی استفاده می کنند که بر اساس عملکرد اصلی یکسان هستند. در این الگو پلتفرم برنامه باید با تکامل مداوم سخت افزار و نرم افزار مقابله کند. پلت فرم برنامه باید قابل حمل، توسعه پذیر و قابل انطباق باشد تا امکان ادغام آسان فناوری های نوظهور را فراهم کند. برنامه های کاربردی در دامنه باید از پلتفرم های مختلف اما مشابه پشتیبانی کنند. برنامه‌ها ممکن است به گروه‌هایی دسته‌بندی شوند که از هسته عملکردی یکسانی به روش‌های مختلف استفاده می‌کنند، که نیاز به پلتفرم برنامه زیربنایی برای تقلید از استانداردهای موجود دارد. هسته عملکردی پلتفرم برنامه باید به یک جزء با حداقل حجم حافظه و سرویس‌های که تا حد ممکن قدرت پردازش کمتری مصرف می‌کنند، جدا شود.

- کپسوله کردن سرویس‌ها اساسی در یک جزء میکروکرنل که با سایر اجزا ارتباط برقرار می کند. حفظ منابع در سراسر سیستم و ایجاد رابط هایی که سایر اجزا را قادر می سازد به عملکرد آن دسترسی داشته باشند.
- سرورهای خارجی دید خود را نسبت به میکروکرنل پیاده سازی می کنند. یک سیستم میکروکرنل ممکن است به عنوان یک پلتفرم کاربردی در نظر گرفته شود که سایر پلتفرم های برنامه را ادغام می کند. مشتریان با استفاده از امکانات ارتباطی ارائه شده توسط میکروکرنل با سرورهای خارجی ارتباط برقرار می کنند.

اجزای اصلی معماری پیشنهادی عبارتند از:

- هسته میکروکرنل:

سرویس‌ها مرکزی (مانند امکانات ارتباطی یا مدیریت منابع) را پیاده سازی می کند. بسیاری از وابستگی های خاص سیستم در میکروکرنل کپسوله می شوند (مثلاً قطعات وابسته به سخت افزار).

- سرورهای داخلی:

افزونه های میکروکرنل (فقط توسط جزء میکروکرنل قابل دسترسی است). میکروکرنل عملکرد سرورهای داخلی را از طریق درخواست های سرویس فراخوانی می کند. برخی از وابستگی ها را به سیستم سخت افزاری یا نرم افزاری زیر پوشش می دهد. اهداف طراحی این است که میکروکرنل را تا حد امکان کوچک نگه داشت تا نیازهای حافظه کاهش یابد. مکانیزم هایی را ارائه می دهد که به سرعت اجرا می شوند تا زمان اجرای سرویس را کاهش دهند.

- سرورهای خارجی:

از میکروکرنل برای پیاده سازی نمای خود از دامنه برنامه زیرین استفاده می کند. آنها درخواست های سرویس را از برنامه های مشتری با استفاده از امکانات ارتباطی ارائه شده توسط میکروکرنل دریافت می کنند، این درخواست ها را تفسیر می کنند، سرویس‌ها مناسب را اجرا می کنند و نتایج را به مشتریان خود برمی گرداند. به عبارتی سرورهاس خارجی Utility Subsystems به عنوان ماژول های پلاگین هستند که سرویس‌ها خاصی را پیاده سازی می کنند و می توانند به صورت پویا بر اساس وضعیت سیستم اضافه یا پیکربندی مجدد شوند.

- کلاینت:

یک برنامه کاربردی را نشان می دهد. اگر فقط به رابط های برنامه نویسی ارائه شده توسط سرور خارجی دسترسی داشته باشد. در هر ارتباطی در کد مشتری هاردکد می شود.

- آداپتور:

محافظت از مشتریان در برابر وابستگی های مستقیم از طریق فضای آدرس مشتری. امکان پیکربندی مجدد در زمان اجرا را فراهم می کند. سرویس‌ها می توانند به صورت پویا بر اساس وضعیت سیستم اضافه یا پیکربندی مجدد شوند.

- Facade :

از آنجایی که فرایندهای پردازشی در بین زیرسیستم پیچیده هستند، این فرایندها را از طریق یک نما یا Facade روی زیرسیستم ها اجرا خواهیم کرد. چون نیاز به داشتن یک رابط محدود اما ساده برای یک زیرسیستم پیچیده داریم، از الگوی الگوی استفاده می‌کنیم زمانی که می خواهیم یک زیرسیستم را به صورت لایه ای ساختار دهید از Facade استفاده کرد..

- وضعیت:

وضعیت کلی سیستم را مشخص و آن را به میکروکرنل اعلام تا بر اساس آن وضعیت پیکربندی مجدد شود. ایده الگوی State این است که، در هر لحظه، تعداد محدودی از حالت‌های سیستم وجود دارد که یک برنامه می‌تواند در آنها قرار گیرد. در هر حالت منحصربه‌فرد، برنامه رفتار متفاوتی دارد و می‌توان برنامه را فوراً از یک حالت به حالت دیگر تغییر داد.

۲-۲ ساختار معماری پیشنهادی

برای معماری پیشنهادی این سیستم از ترکیب الگوها بهره میبریم:

۱. الگوی میکروکرنل به عنوان سیستم اصلی مدیریت ارتباطات که مدیریت پیکربندی را تسهیل می کند. این الگو امکان تعامل بین کلاینت، سیستم هسته و ماژول های مختلف پلاگین (زیرسیستم‌های utility) را تسهیل می کند. همچنین این الگو افزودن، حذف و پیکربندی پویا زیرسیستم ها را فراهم می کند.
۲. الگوی مشاهده گر زیر سیستم ها را به صورت پویا مانیتور می کند.
۳. الگوی وضعیت به زیرسیستم های مشتری اجازه می دهد تا به صورت پویا با تغییرات حالت در سیستم اصلی با زیرسیستم های ابزار سازگار شوند. اگر تغییر حالت رخ دهد، سیستم اصلی به همه Observer اطلاع می دهد.
٤. الگوی استراتژی ارائه استراتژی‌های مختلف برای پیکربندی زیر سیستم‌ها را در زمان اجرا، فراهم می‌کند. خانواده ای از الگوریتم ها (استراتژی‌ها) را برای سرویس های مختلف تعریف می‌کند و زیرسیستم های utility را قابل تعویض کرد. این به سیستم اجازه می دهد تا به صورت پویا بین استراتژی های مختلف بر اساس وضعیت سیستم جابجا شود.
٥. الگوی آداپتور اطمینان حاصل میکند که زیرسیستم های مختلف علیرغم تفاوت‌های رابط می توانند به طور یکپارچه با هم کار کنند. به سیستم اجازه می دهد تا بسیار سازگار و قابل تنظیم مجدد باشد و نیازهای زیرسیستم های کاربردی توزیع شده و زیرسیستم های مشتری را برطرف کند.
٦. الگوی Façade سارائه یک رابط یکپارچه برای مجموعه ای از اینترفیس‌های زیرسیستم های utility، استفاده از سیستم‌ها را برای پاسخ به درخواست‌های مشتری آسان تر می‌کند.
۷. الگوی زنجیره مسئولیت، زیرسیستم های utility را برای پاسخ به درخواست مشتری در یک زنجیره پیوند می دهد.

۲-۳ نمودار مولفه

مولفه ها عبارتند از:

۱. مولفه میکروکرنل: شامل هسته میکروکرنل است سرویس‌ها اساسی مانند ارتباطات، پیکربندی و مدیریت وضعیت را ارائه می دهد. سرویس‌ها و زنجیره سرویس را مدیریت می کند. سرویس‌ها را ثبت و از ثبت خارج می کند. شامل ماشین حالات

۲. State Machine: بخشی از هسته میکروکرنل که وضعیت‌های مختلف سیستم را مدیریت می‌کند و رفتارهای مرتبط با هر وضعیت را تعیین می‌کند.

۳. Strategies: بخشی از هسته میکروکرنل که الگوریتم‌ها و رفتارهای قابل تغییر در زمان اجرا را مدیریت می‌کند.

۴. زیرسیستم های utility: اجزای مدولار هستند که سرویس‌های خاصی را ارائه می دهند. آنها را می توان به صورت پویا اضافه، حذف یا پیکربندی مجدد کرد.

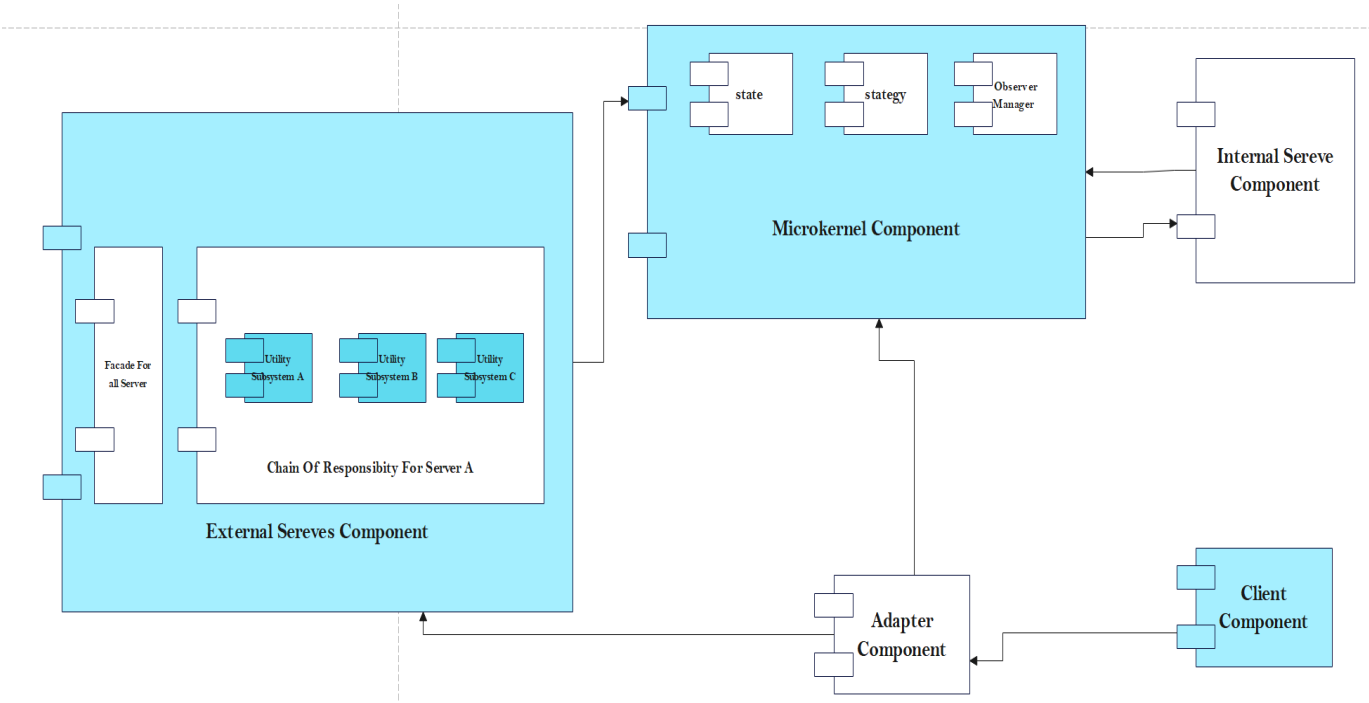
۵. Observer جمع آوری داده ها و نظارت بر وضعیت زیرسیستم های utility را انجام میدهد. همچنین تغییرات مربوطه را به مولفه state اطلاع می دهد.

۶. Adapter: آداپتور درخواست کلاینت ها به یک سرور خارجی متصل می‌کند. به عنوان یک پروکسی که دقیقاً یک سرور خارجی را نشان می دهد. اطمینان حاصل می کند که زیرسیستم های utility با رابط های مختلف می توانند با هم کار کنند.

۷. Façade: برای مجموعه سرورها یک فساد وجود دارد که درخواست ورودی را به سرور یا سروهای مرتبط ارسال می‌کند.

۸. مدیر زنجیره سرویس‌ها: زنجیره‌سازی زیرسیستم‌های utility را مدیریت می‌کند.

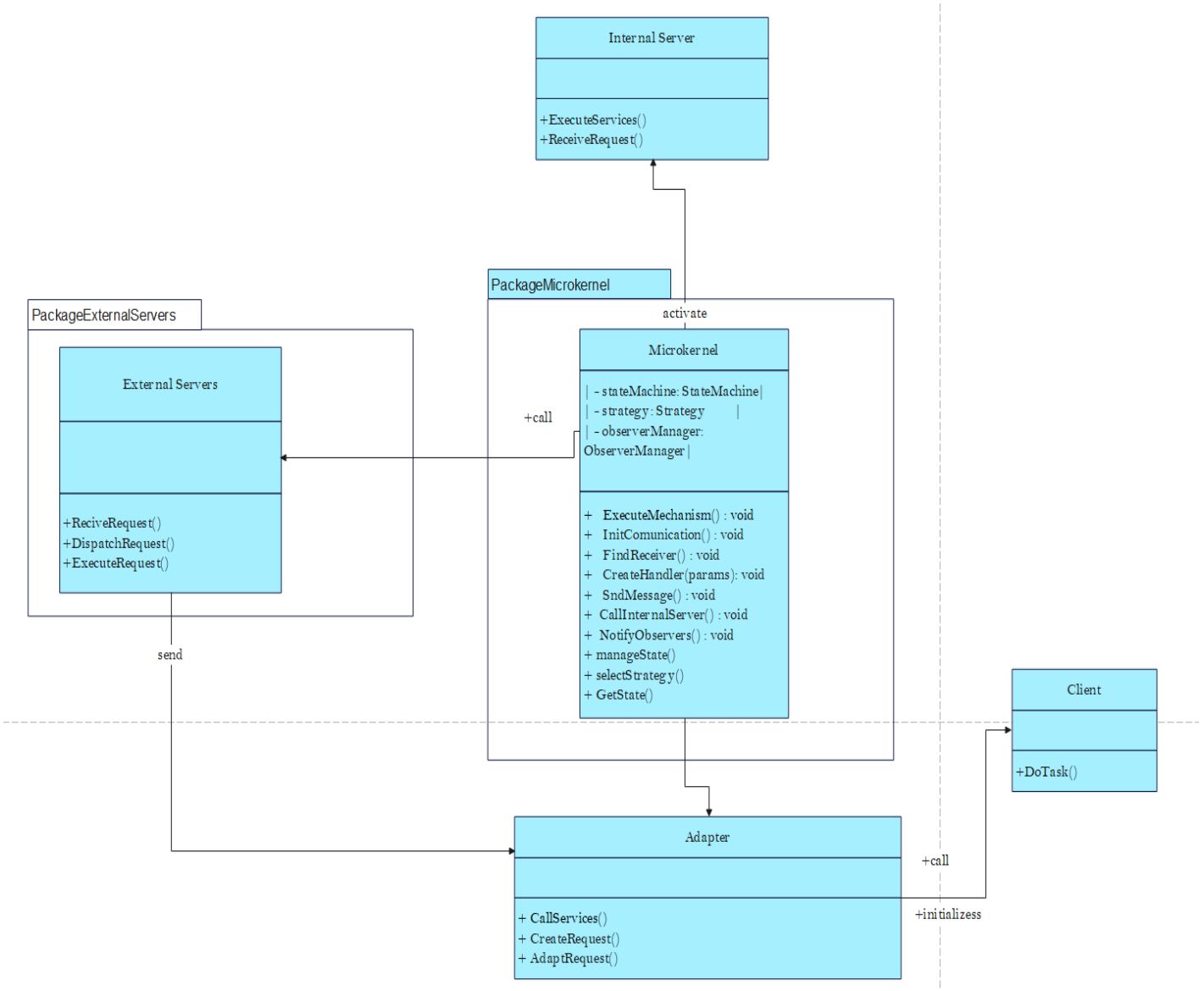
۹. مشتری: سرویس‌های ارائه شده توسط زیرسیستم های utility را مصرف می کنند.



شکل ۷- نمای کلی دیاگرام مولفه الگوی معماری پشنهادی

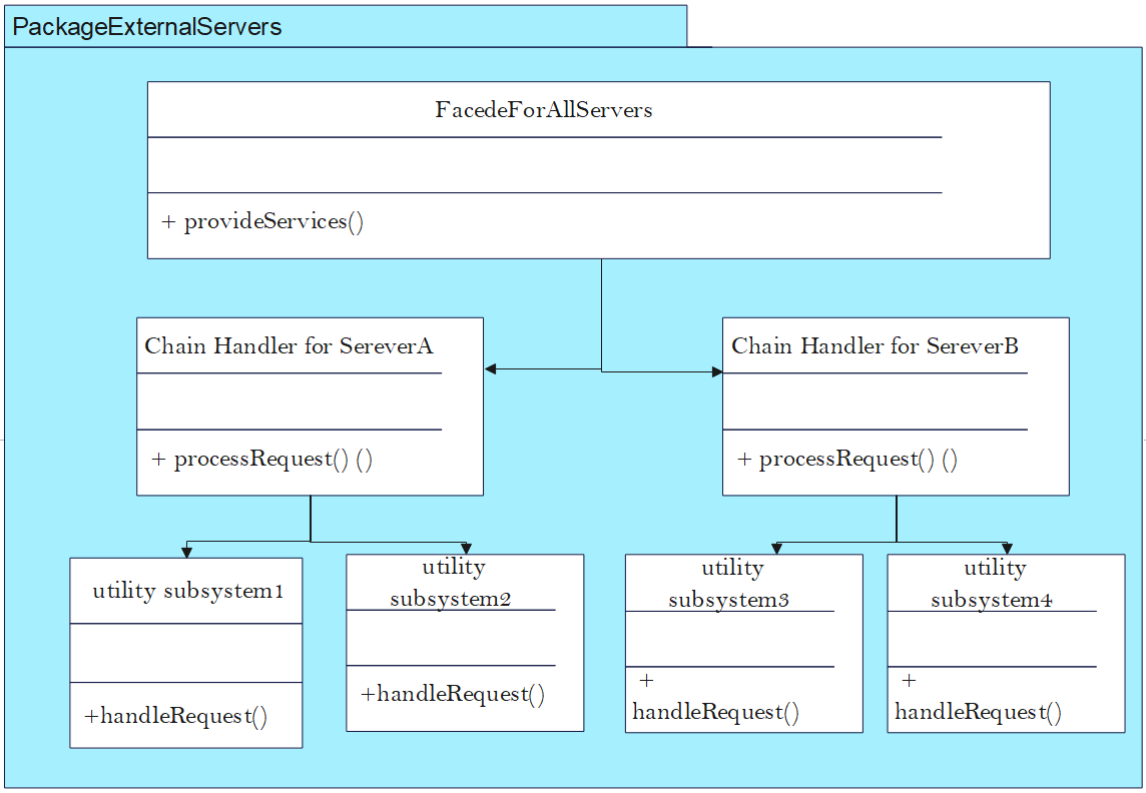
۲-۴ دیاگرام کلاس

کلاس دیاگرام اجزای کلی سیستم:



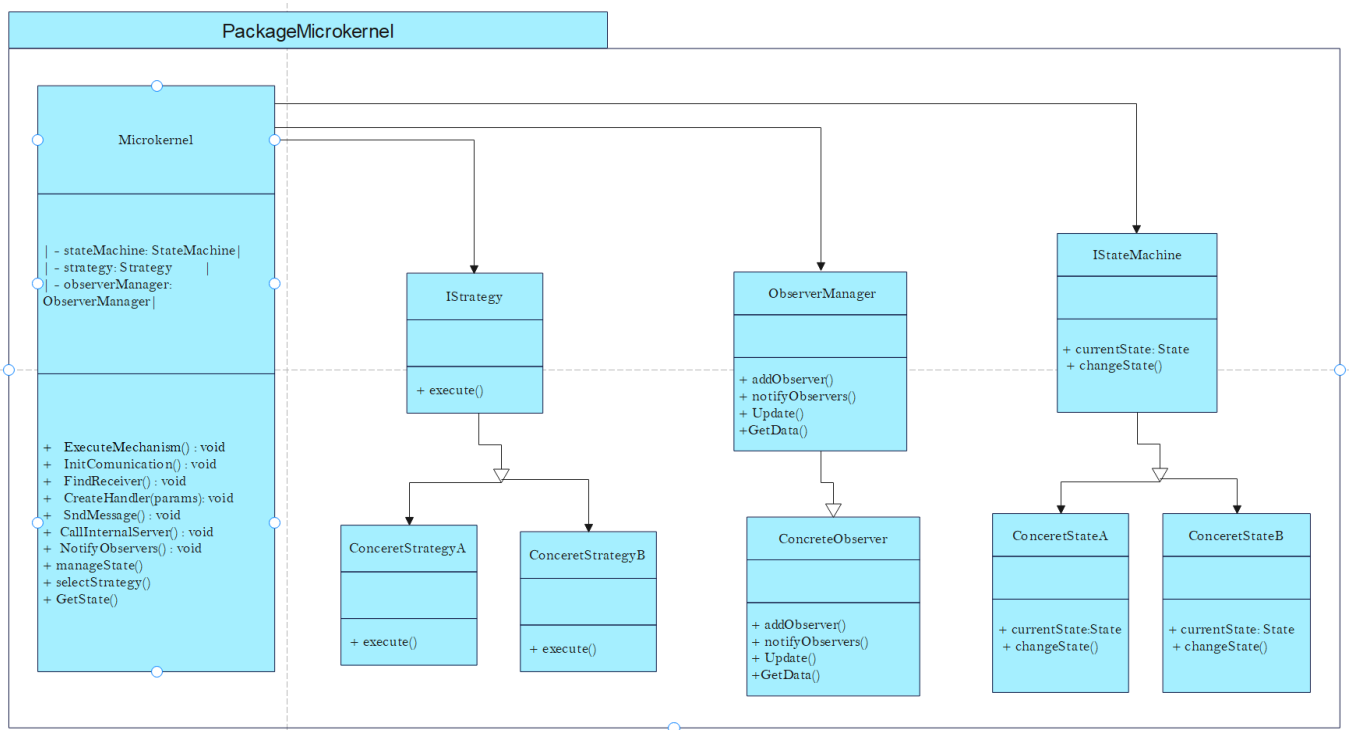
شکل ۷- نمای کلی کلاس دیاگرام الگوی پیشنهادی

محتویات داخل پکیج اکسترنال سرور:



شکل ۸- نمای کلی داخل پکیج اکسترنال سرور الگوی پیشنهادی

محتویات داخل پکیج میکروکرنل



شکل ۹- نمای کلی داخل پکیج میکروکرنل الگوی پیشنهادی

۲-۵ رفتار معماری پیشنهادی

سناریوی۱ : کلاینت سرویسی را فراخوانی می کند.

هنگامی که یک کلاینت با سرویس سرور خارجی خود تماس می گیرد، رفتار زیر را نشان می دهد.

۱. در یک نقطه مشخص از کنترل جریان، کلاینت با فراخوانی آداپتور، سرویسی را از یک سرور خارجی درخواست می کند.

```
ClientSubsystem -> Adapter: requestService ()
```

۲. آداپتور یک درخواست ایجاد می کند و از میکروکرنل می خواهد که یک پیوند ارتباطی با سرور خارجی داشته باشد.

```
Adapter -> MicrokernelCore: initComunication ()
```

۳. میکروکرنل وضعیت فعلی سیستم را تعیین و بر اساس آن یک استرتژی را انتخاب می کند.

```
MicrokernelCore -> StateMachine: getCurrentState()
StateMachine -> MicrokernelCore: currentState
MicrokernelCore -> Strategy: execute(currentState)
Strategy -> MicrokernelCore: strategyResult
```

۴. میکروکرنل آدرس فیزیکی سرور خارجی را تعیین می کند و آن را به آداپتور برمی گرداند.

۵. پس از بازیابی این اطلاعات، آداپتور یک پیوند ارتباطی مستقیم با سرور خارجی ایجاد می کند.

۶. آداپتور درخواست را با استفاده از یک فراخوانی از راه دور به سرور خارجی ارسال می کند.

۷. فساد در سرور خارجی درخواست را دریافت می کند، پیام را باز می کند و کار را به یکی از سرورهای های خود محول می کند..

```
Adapter -> Fadcade: ReceiveRequest ()
```

۸. هندلر زنجیره درخواست را از فساد دریافت و میان زیر سیستم های در طول زنجیره عبور میدهد و در پایان پاسخ مناسب را به فساد بر گرداند.

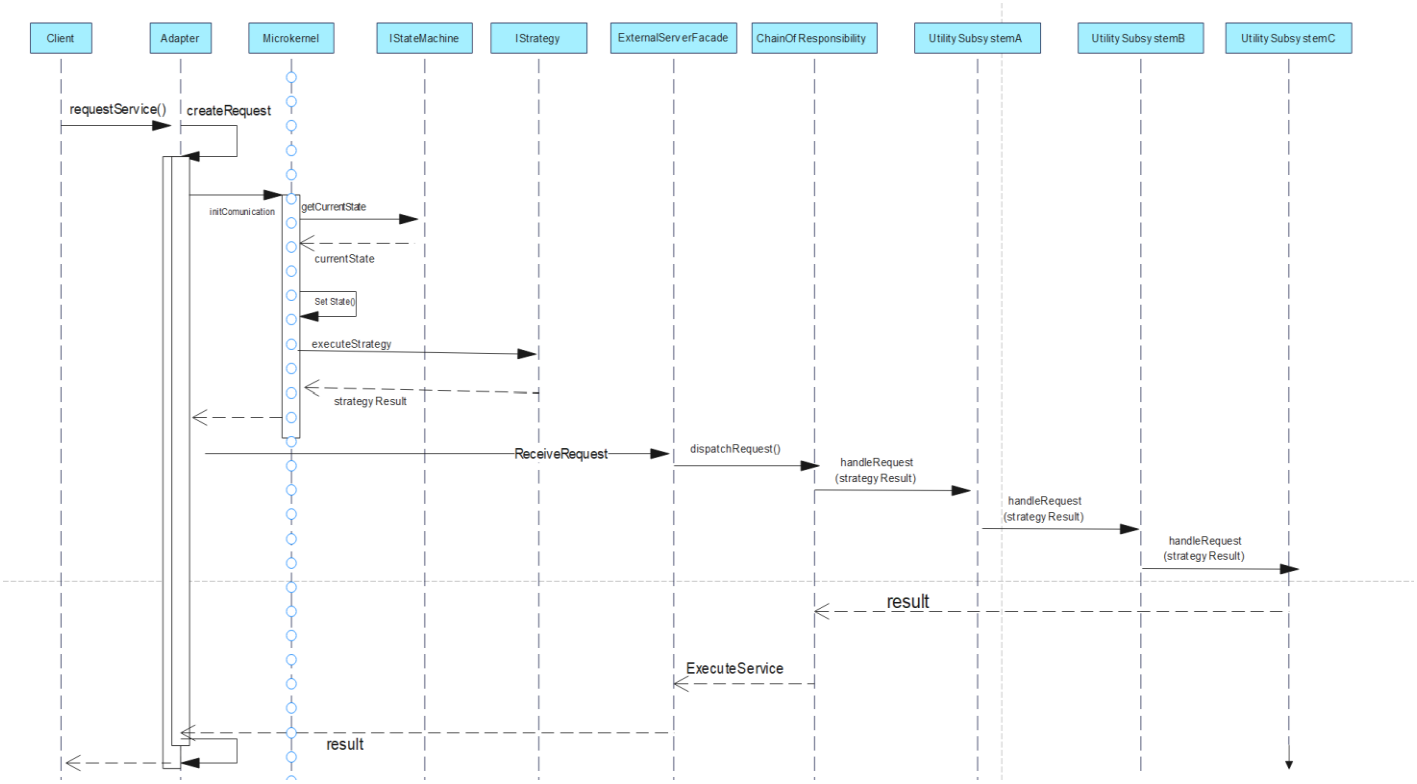
```
Fadcade -> ChainOfResponsibility: dispatchRequest()
ChainOfResponsibility -> UtilitySubsystemA: handleRequest(strategyResult)
UtilitySubsystemA -> UtilitySubsystemB: handleRequest(strategyResult)
UtilitySubsystemB -> UtilitySubsystemC: handleRequest(strategyResult)
UtilitySubsystemC -> ChainOfResponsibility: result
ChainOfResponsibility -> Façade: result
```

۹. فساد نیز پس از تکمیل سرویس درخواستی، تمام نتایج و اطلاعات وضعیت را به آداپتور ارسال می کند.

```
Façade -> Adapter: result
```

۱۰. آداپتور به مشتری باز می گردد، که به نوبه خود به جریان کنترل خود ادامه می دهد.

```
Adapter -> ClientSubsystem: result
```



شکل ۱۰- نمای کلی دیاگرام توالی ۱ الگوی پیشنهادی

سناریوی ۲: حالت سیستم تغییر می کند و نیاز به پیکربندی مجدد دارد.

حالت سیستم بر اساس یک رویداد تغییر می کند و میکروکرنل استراتژی را مجدد پیکربندی می کند تا بتواند درخواست های بعدی را بر اساس وضعیت جدید پاسخ بدهد.

۱. مشاهده گر پیغام Notify() را به میکروکرنل ارسال می کند تا او را از رویداد مطلع کند.

Observer -> c: notify(event)

۲. میکروکرنل وضعیت را بررسی و در صورت لزوم state را تغییر میدهد و استراتژی مناسب را اتخاذ می کند.

MicrokernelCore -> StateMachine: changeState(event)
StateMachine -> MicrokernelCore: newState
MicrokernelCore -> Strategy: execute(currentState)
Strategy -> MicrokernelCore: strategyResult

۳. میکروکرنل پیغام به روزرسانی را به فساد سرورهای خارجی ارسال می کند.

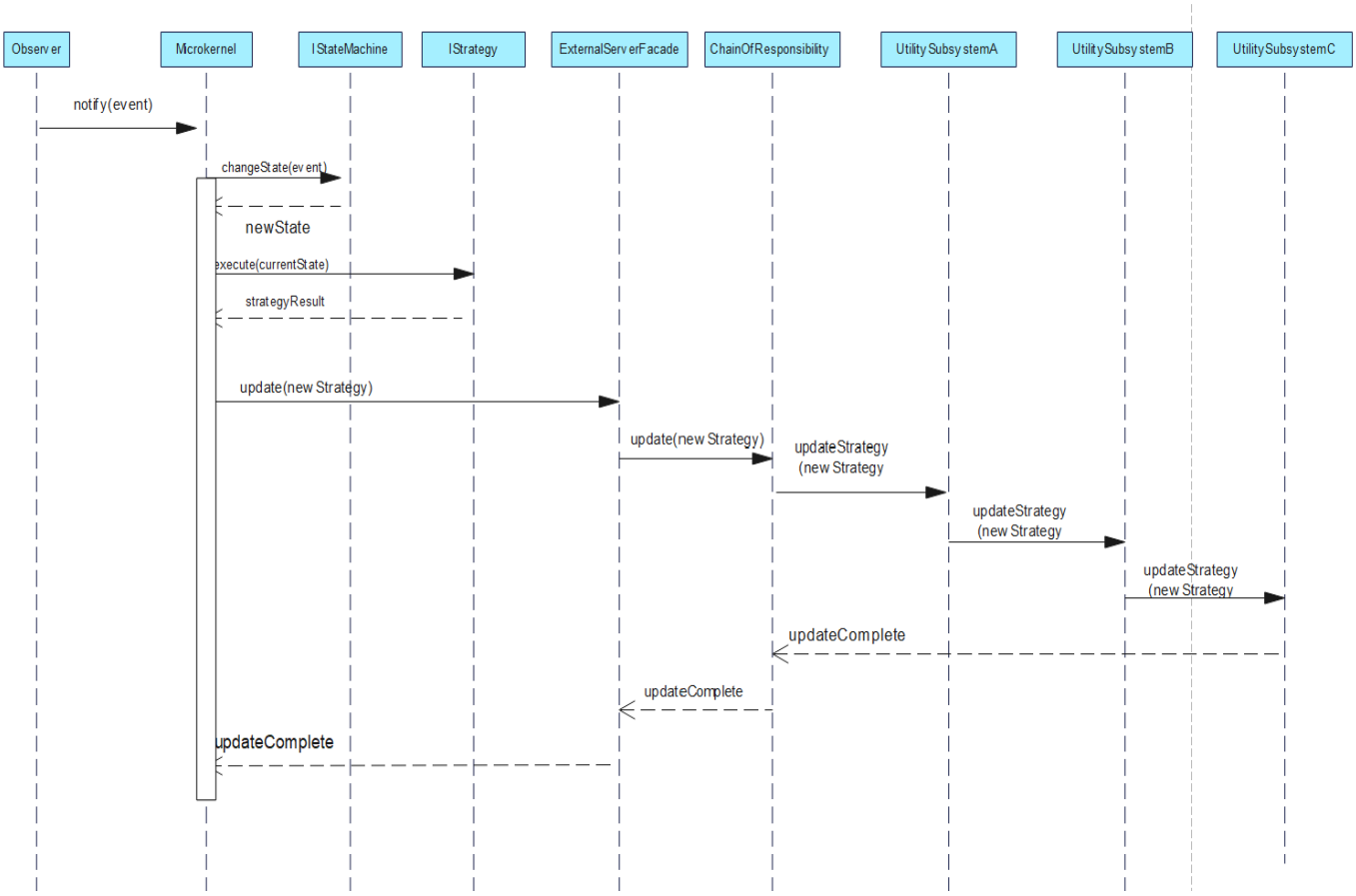
MicrokernelCore -> Façade: update(newStrategy)

٤. فساد پس از دریافت پیغام به هندلر زنجیره سرورها پیغام را ارسال میکند و در نهایت این پیغام در طول زنجیره عبور کرده و نتیجه به هنرلر بر می گردد.

Façade -> ChainOfResponsibility: update(newStrategy)
ChainOfResponsibility -> UtilitySubsystemA: updateStrategy(newStrategy)
UtilitySubsystemA -> UtilitySubsystemB: updateStrategy(newStrategy)
UtilitySubsystemB -> UtilitySubsystemC: updateStrategy(newStrategy)
UtilitySubsystemC -> ChainOfResponsibility: updateComplete
ChainOfResponsibility -> Facade: updateComplete

٥. هندلر زنجیره پاسخ را به فساد بر گرداند.

Facade -> MicrokernelCore: updateComplete



شکل ۱۰- نمای کلی دیاگرام توالی ۲ الگوی پیشنهادی

۲-۶ مزایا معماری پیشنهادی

۱. مدولار بودن و مقیاس پذیری:هنگامی که یک ماشین جدید به شبکه اضافه می کنیم، مقیاس کردن سیستم Microkernel به پیکربندی جدید آسان است. Microkernel با کوچک نگه داشتن سیستم هسته و گسترش عملکرد از طریق پلاگین ها یا زیرسیستم های کاربردی، ماژولار بودن را ارتقا می دهد. این امکان مقیاس‌پذیری آسان‌تر را فراهم می‌کند زیرا ویژگی‌ها یا سرویس‌ها جدید می‌توانند بدون تغییر قابل توجهی در سیستم اصلی اضافه شوند.

۲. سازگاری پویا: استفاده از الگوهایی مانند زنجیره مسئولیت، مشاهده‌گر، وضعیت و استراتژی، امکان انطباق پویا را بر اساس تغییر وضعیت‌ها یا الزامات سیستم ممکن می‌سازد. این باعث افزایش انعطاف پذیری سیستم می شود.

۳. قابلیت اطمینان: این امکان را می دهد یک سرور را روی بیش از یک دستگاه اجرا کنیم و در دسترس بودن را افزایش دهیم.

۴. تحمل خطا ممکن است به راحتی پشتیبانی شود زیرا سیستم های توزیع شده به ما این امکان می دهند خرابی ها را از کاربر پنهان کنیم.

۵. پیکربندی و پیکربندی مجدد: با استفاده از الگوهایی مانند استراتژی، سیستم در زمان اجرا بسیار قابل تنظیم و تنظیم مجدد می شود. این امکان سفارشی سازی آسان سرویس‌ها را برای برآورده کردن نیازهای مشتری خاص بدون خرابی فراهم می کند.

۶. شفافیت تمام جزئیات ارتباط بین فرآیندی با سرورها توسط آداپتورها و میکروکرنل از مشتریان پنهان می شود.

۷. همکاری شده: استفاده از الگوهایی مانند زنجیره مسدولیت همکاری یکپارچه بین زیرسیستم های ابزار توزیع شده را تسهیل می کند و ارائه سرویس‌ها کارآمد را در یک محیط توزیع شده امکان پذیر می کند.

۲-۷ معایب معماری پیشنهادی

۱. پیچیدگی: پیاده سازی الگوهای متعدد می تواند پیچیدگی سیستم را افزایش دهد و درک، نگهداری و اشکال زدایی آن را دشوارتر کند. این پیچیدگی همچنین ممکن است بر عملکرد تأثیر بگذارد.

۲. سربار: برخی از الگوها، مانند Chain of Responsibility و Observer، سربار را به دلیل لایه های اضافی غیرمستقیم یا مدیریت رویداد معرفی می کنند. این سربار می تواند بر عملکرد سیستم تأثیر بگذارد، به ویژه در سناریوهای با توان عملیاتی بالا.

۳. مدیریت وابستگی: از آنجایی که سیستم به الگوها و زیرسیستم های متعدد متکی است، مدیریت وابستگی ها بین آنها بسیار مهم می شود. تغییرات در یک زیرسیستم یا الگو ممکن است اثرات منتشرشونده بر سایرین داشته باشد که منجر به چالش های بالقوه یکپارچه سازی شود.

به طور کلی، در حالی که الگوی پیشنهادی مزایای متعددی را از نظر انعطاف‌پذیری، پیکربندی و مقیاس‌پذیری ارائه می‌دهد، باید توجه دقیقی به مدیریت پیچیدگی و سربار برای اطمینان از اجرای موفقیت‌آمیز داده شود.

۳– شرح مساله دوم

idiom ها الگوهای سطح پایین زبان هستند و توصیف چگونگی پیاده سازی بخشی هایی خاص از کامپوننت یا رابطه بین آنها با ویژگی های زبان خاص است. idiom ها چگونگی حل مشکلات پیاده سازی در زبان هستند. در ادامه دو idiom با نام های Exception و Enumeration and Collection مربوط به زبان برنامه سازی جاوا معرفی خواهد شد.

۳–۱ تعریف اصطلاح *NestedException*

اصطلاح "NestedException Idiom" برای رسیدگی به شرایطی استفاده می‌شود که در آن یک استثنا در حین رسیدگی به یک استثنا دیگر ایجاد شود. به عبارتی برای کپسوله کردن یک استثنا اصلی در یک استثنا جدید استفاده می شود. این روش این اجازه می دهد تا استثنای اصلی حفظ شده و ارجاع داده شود و ردیابی علت اصلی یک خطا آسان‌تر شود.

۳–۲ هدف

برای اطمینان از اینکه استثنائات در سطح مناسبی در برنامه مورد بررسی قرار می‌گیرند، جایی که بازیابی یا پاسخ معنی‌دار ممکن است رخ دهد.

۳–۳ انگیزه

در مدیریت استثناها، درک زمینه و توالی رویدادهایی که منجر به خطا شده است، اغلب مفید است. با تودرتو کردن استثناها، توسعه‌دهندگان می‌توانند زنجیره روشنی از رویدادها را حفظ کنند و اشکال‌زدایی را مؤثرتر کنند.

۳–۴ راهکار

راه حل شامل ایجاد یک استثنا است که می تواند شامل یک استثنا دیگر (استثنای تودرتو) باشد. به این ترتیب، اگر یک استثنا در هنگام مدیریت یک استثنا دیگر رخ دهد، استثنای جدید می تواند به استثنای اصلی ارجاع دهد و ردپای پشته و زمینه را حفظ کند.

ایده کلی این است که استثناها باید شامل ارجاعاتی به استثنایی باشند که باعث ایجاد آنها شده است. هنگامی که یک لایه یک استثنا را می گیرد(catches) و یک استثنای جدید را پرتاب می کند(throws)، استثنای جدید باید شامل ارجاع به "علت" قبلی باشد.
این کد اصطلاح NestedException را در جاوا در متد startElement تجزیه کننده SAX نشان می‌دهد. یک عنصر XML را پردازش می‌کند و استثناهای خاص را با تودرتو کردن آنها در SAXException مدیریت می کند و جزئیات استثنای اصلی را حفظ می‌کند.

```
public void startElement (String name, AttributeList attrs)
        throws SAXException {
        try {
        if (name.equals("TVSCHEDULE")) {
                this.ParseStartSchedule(attrs);
                return;
                }

        // [..endless if/else deleted...]
        return;
        }
        catch (IOException e) {
          throw new SAXException("I/O error: "+e.toString(), e);
          }
        catch (TVScheduleParseException e) {
          throw new SAXException("TVScheduleParseError"+e.toString(), e);
          }
        }
```

شکل کلی این دستور به شکل زیر است:

```
try {
  // Code that may throw an exception
} catch (SomeException e1) {
  try {
```

```
        // Code that may throw another exception
    } catch (AnotherException e2) {
        throw new NestedException("An error occurred while handling another exception", e1, e2);
    }
}
```

۳–۵ تعریف اصطلاح *ImmutableCollection*

گاهی لازم است از انجام اصلاحات روی یک مجموعه(collection) در زمانی که مجموعه در حال "پیمایش" است، توسط یک شی Enumeration پشتیبانی شود. یا ممکن است چندین رشته بخواهند به طور همزمان روی مجموعه پیمایش انجام بدهند و آن را اصلاح کنند. از این رو از ساختار داده ای استفاده می‌کنیم که تغییر ناپذیر است.

"immutablecollection" مجموعه تغییرناپذیر نوعی مجموعه است که بعد از ایجاد حالت آن قابل تغییر نیست. این بدان معناست که پس از مقداردهی اولیه مجموعه، نمی توان عناصر را اضافه، حذف یا تغییر داد.

اشیایی که حاوی ImmutableCollection هستند، هر زمان که محتوای مجموعه را تغییر دهند، مرجع خود را به ImmutableCollection با نسخه اصلاح شده جایگزین می کنند. این محاسبه و تخصیص باید اتمی باشد تا از تغییرات همزمان که باعث از دست رفتن به‌روزرسانی‌ها می‌شود جلوگیری شود، و بنابراین باید در یک بلوک همگام‌سازی انجام شود.

۳–۶ هدف

هدف اصلی مجموعه‌های تغییرناپذیر این است که اطمینان حاصل شود که ساختار داده در طول چرخه عمر خود ثابت می‌ماند، که استدلال در مورد کد را ساده می‌کند و ایمنی نخ (thread) را افزایش می‌دهد.

۳–۷ انگیزه

مجموعه های تغییرناپذیر با انگیزه نیاز به سادگی، قابلیت اطمینان و سهولت نگهداری در طراحی نرم افزار ایجاد می شوند. آنها مسائل مربوط به تغییرات همزمان را حذف می کنند و آنها را برای محیط های چند نخی ایده آل می کنند.

۳–۸ راهکار

راه حل شامل طراحی مجموعه هایی است که روش هایی برای اصلاح ارائه نمی دهند. درعوض، عملیاتی که مجموعه را اصلاح می‌کند، مجموعه‌های جدید را با داده‌های به‌روز شده برمی‌گرداند و مجموعه اصلی را بدون تغییر می‌گذارد.

در مثال زیر با استفاده از مجموعه غیرقابل تغییر (ImmutableList) به روشی ایمن روی یک نخ اصلاحات انجام میگیرد. با استفاده از این idiom جاوا تضمین میکند که خواندن و نوشتن یک شی مرجع در یک عملیات اتمی انجام میشود.

```
private volatile ImmutableList list = ImmutableList.EMPTY;
private final Object listLock = new Object();

public Enumeration enumerateChildren() {
    return list.elements();
}
public void addChild( Object child ) {
    synchronized(listLock) {
        list = list.add(child);
    }
}
```

۴– منابع

[2] Ramsin, R., 2023. Lecture1 to 8 in Patterns in Software Engineering course : GoF Design Patterns

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. Pattern-Oriented Software Architecture - Volume 1: A System of Patterns. Wiley Publishing.

[3] http://wiki.c2.com/?ImmutableCollection

[4]