دانشکده مهندسی کامپیوتر

تمرین دوم درس الگوها در مهندسی نرم افزار

استاد:

دکتر رامسین

تهیه کننده:

امید یعقوبی سامانی

۴۰۱۲۰۵۳۴۸

بهار ۱۴۰۱

فهرست مطالب

فهرست شکل ها و جداول

پیشنهاد معماری برای مسئله مورد نظر براساس الگوهای GoF و GoV:

تعریف مسئله:

در یک سیستم توزیع شده تعدادی زیر سیستم در حال تعامل به روش پیچیده ای با یک زیرسیستم مشترک هستند. تامین امنیت برای زیرسیستم مشترک مهم شده است. روش های امنیتی پیچیده هستند و بار سنگینی به منابع سیستمی تحمیل می کنند برای همین همیشه در یک سطح به سیستم اعمال نمی شوند. سطوح امنیتی بستگی به تنظیمات مدیریتی و سطوح مخاطره دارد. باید سیستمی طراحی شود که همیشه امنیت کافی را برای زیرسیستم مشترک تامین کند و ضمنا در منابع سیستمی هم صرفه جویی شود.

نیازمندی های اصلی:

- باید سیستم توزیع شده باشد
- باید سیستم تعاملات پیچیده را اداره کند
- باید بتوان سطوح مختلف امنیتی اعمال کرد
- باید بتوان سطوح امنیتی مختلف را براساس تنظیمات مدیریتی یا سطوح مخاطره اعمال کرد
- باید علاوه بر تامین امنیت در منابع صرفه جویی صورت بگیرد

راه حل پیشنهادی برای نیازمندی ها:

توزیع شدگی:

در سیستم های توزیع شد هر بخش سیستم که می تواند خودش یک زیر سیستم باشد روی یک گره در شبکه می نشیند و با یکدیگر تعامل می کنند. کامپوننت ها باید کمترین coupling را به یکدیگر داشته باشند. دسترسی ها به صورت راه دور است و همه چیز هم باید location transparent باشد یعنی کلاینت هایی که قرار است از سرور، سرویسی بگیرند باید سرور را محلی ببینند و همچنین سروری که پاسخ می دهد هم باید کلاینت را محلی ببیند. دغدغه وابستگی ها به پلتفرم، زیر ساخت شبکه و نحوه ارتباط نباید روی دوش کلاینت ها و سرور ها باشد و در این محیط نباید هیچ گونه وابستگی به توزیع شدگی وجود داشته باشد.

معماری پیشنهادی براساس معماری های توزیع شده مطرح، استفاده از معماری Broker است. 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. در این معماری دغدغه ارتباطات که شامل یافتن سرور و کلاینت ها است را Broker به عهده دارد. همچنین در سمت کلاینت Client-Side Proxy داریم که نقش سرور را برای کلاینت بازی می کند و علاوه بر ارسال درخواست به Broker عملیات بسته بندی داده در محیط توزیع شده و بازگردانی آن را انجام می دهد. در سمت سرور هم یک Server-Side Proxy داریم که نقش کلاینت را برای سرور دارد و کارهای مشابه Client-Side Proxy را انجام می دهد.

مدیریت تعاملات پیچیده:

در مسئله، تعاملات پیچیده است و باید دغدغه تعاملات از روی دوش کلاینت و سرور برداشته شود که Broker این وظیفه را به عهده می گیرد. سرور خود و سرویس هایش را در Broker ثبت می کند و Broker هم اینترفیس سرویس هایی که وجود دارند را برای کلاینت ها افشا می کند. کلاینت ها تنها کافی است برای دریافت سرویس مورد نظر خود به Broker پیغام بدهند. خود Broker مکان سرور ها را یافته و درخواست را برای سرویس مورد نظر به آنها ارسال، پاسخ را دریافت و با یافتن کلاینت پاسخ را برای کلاینت ارسال می کند. کلاینت و سرور هیچ گونه دغدغه تعامل ندارند و کلاینت فکر می کند درخواست را به طور محلی به سرور می دهد و سرور هم گمان می کند پاسخ برای کلاینت محلی ارسال می شود.

اعمال سطوح مختلف امنیتی:

برای اینکه بتوان سطوح مختلف امنیتی را روی سیستم اعمال کرد از چندین الگوی مختلف استفاده می شود. اولا برای اینکه مسئله مربوط به مسائل امنیتی سیستم است باید همه چیز مرکزی باشد و از یک نقطه اعمال شود. برای اینکار از الگوی Proxy استفاده می شود. یک Proxy داریم که درخواست ها را گرفته و سطوح امنیتی مختلف را روی آنها اعمال و در صورتی که تمام سطوح رعایت شده بودند درخواست را به Broker ارسال می کند. در صورتی یکی از موارد انجام نپذیرفت پاسخ مناسب برای کلاینت درخواست دهنده ارسال خواهد شد. برای اعمال سطوح مختلف درProxy با استفاده از الگوی Strategy درخواست به یک استراتژی امنیتی که توسط مدیریت سیستم یا براساس سطوح مخاطره تعیین شده ارسال می شود.

در این استراتژی الگوریتم اعمال سطوح مختلف پیاده سازی می شود. این کار به کمک الگوی Chain of Responsibility انجام می شود. موارد مختلف امنیتی به صورت ریز دانه تعریف می شوند و با کمک این الگو زنجیره ای از آنها ایجاد می گردد. می توان با ریزدانه کردن سرویس های کنترل امنیت و با کمک الگو انواع مختلف استراتژی های امنیتی را طراحی نمود. با حرکت در زنجیره درخواست در هر کدام از دانه های زنجیره رد و در صورتی که موفق به گذر از آن شد یعنی آن بخش امنیتی مورد نظر را داشته است ولی اگر موفق نشد پاسخ مناسب برگشته و زنجیره تمام می شود و ادامه نمی یابد و پاسخ به کلاینت برگشت داده می شود. یک درخواست تنها در صورت گذر از تمام عناصر زنجیره می تواند به Broker وارد شود. در اینجا با قرار دادن پروکسی دغدغه اعمال سطوح امنیتی را از دو Broker برداشته و Broker تنها باید کار های مورد نظر خود را انجام بدهد.

سطوح مخاطره:

سطوح مختلف استراتژی امنیت یا توسط مدیریت اعمال می شود یعنی یک نمونه از کلاس های استراتژی را ساخته و به پروکسی می دهد یا باید توسط ویژگی های مختلف سیستم و براساس میزان مخاطره تعیین شود. در اینجا بعد از تعیین سطح مخاطره باید استراتژی مناسب بر آن اساس به پروکسی داده شود.

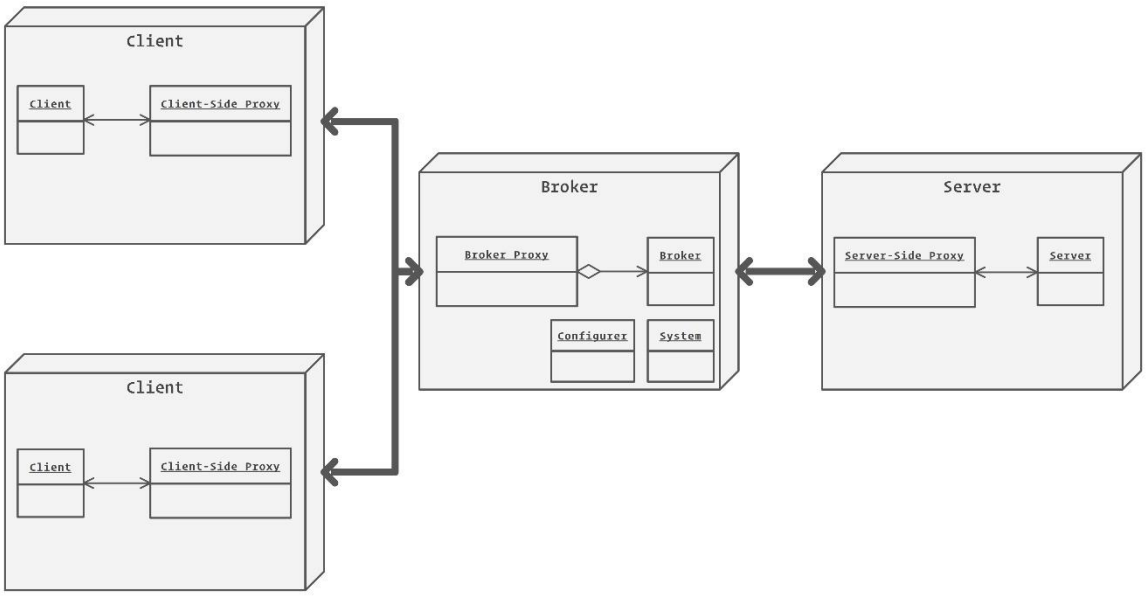
برای بررسی سطوح مخاطره از الگوی Observer و State استفاده شده است. در سیستم تعدادی State مختلف داریم که براساس آنها RiskAssessment می تواند سطح امنیتی مورد نظر را تامین نماید. هر کدام از این State ها Observer یک سری Monitor در سیستم هستند که اطلاعات مختلف را در سیستم جمع آوری می کنند. مثلا Monitor مربوط به کلاینت می تواند دسترسی های کلاینت های مختلف را Log کند که این کار را می تواند در پروکسی انجام بدهد یا می توان Monitor برای Fault و خطا ها داشت که می تواند رد شدن در زنجیره امنیتی را log کند همچین Monitor برای منابع می توان داشت که میزان مصرف را بررسی کند و در هر زمینه ای که ممکن است در ریسک مهم باشد می توان این موارد را داشت که در بخش های مختلف سیستم وجود خواهند داشت. در این Monitor ها پس از تغییر حالت، این تغییر حالت ها به State مورد نظرش که observer آن است ارسال و State با دریافت حالت Monitor می تواند state را تغییر بدهد. تغییر این state می تواند منجر به تغییر مکانیزم امنیتی در RiskAssessment بشود یا ممکن است شرایط امنیتی تغییری نکند. در صورت تغییر به سیستم مرکزی اطلاع داده می شود تا استراتژی را تغییر و پیکربندی مجدد انجام دهد.

صرفه جویی در منابع:

صرفه جویی در منابع مشابه مورد قبل است و در همان جا تعیین خواهد شد و Monitor مربوط به منابعی که وجود دارد یا حتی Monitor مربوط به کلاینت ها یا هرMonitor دیگری با تغییر حالتش می تواند باعث تغییر حالت امنیتی سیستم شود که به طبع آن باعث می شود در منابع صرفه جویی شود و سیستم دچار کندی در شرایط خاصی نباشد این کار را انجام خواهند داد.

ساختار راه حل:

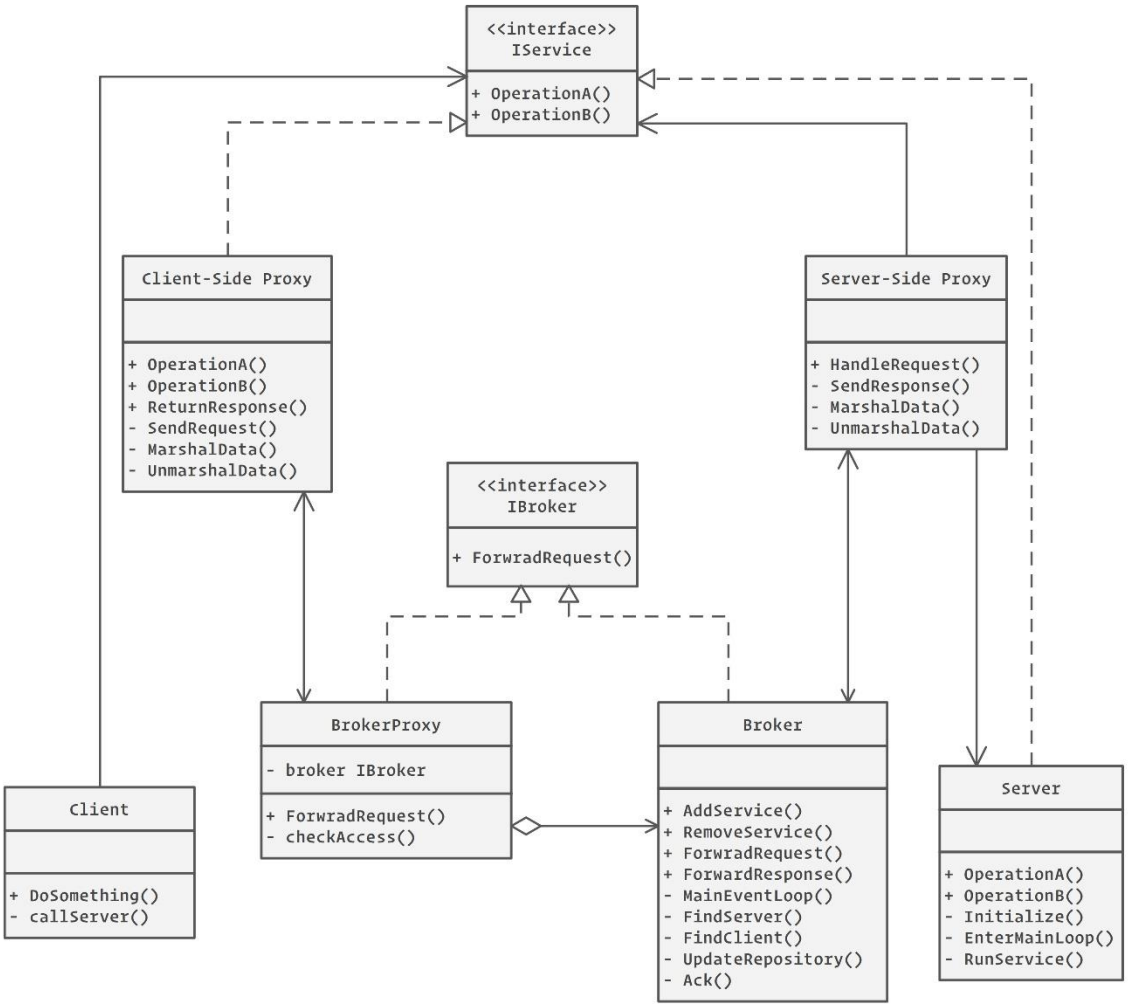
ساختار سطح بالای راه حل و نحوه استقرار مؤلفه های مختلف مطابق شکل ۱ است. هر بخش یک گره شبکه است که در یکی کلاینت یا زیرسیستم در یکی سرور یا زیرسیستم مشترک و در دیگری Broker و BrokerProxy که مسئول تامین امنیت است قرار گرفته اند.



شکل ۱- نمودار استقرار راه حل

در شکل ۲ نمودار کلاس مربوط به الگو مشخص شده است که سطح بالا است و شامل Broker و BrokerProxy و کلاینت و سرور و تعاملات اینها با هم و نحوه تامین امنیت و اعمال سطوح مختلف امنیتی روی سیستم است.

اینترفیس ها، کلاس های پیاده سازی کنند و ارتباطات مختلف آنها در شکل مشخص شده است.

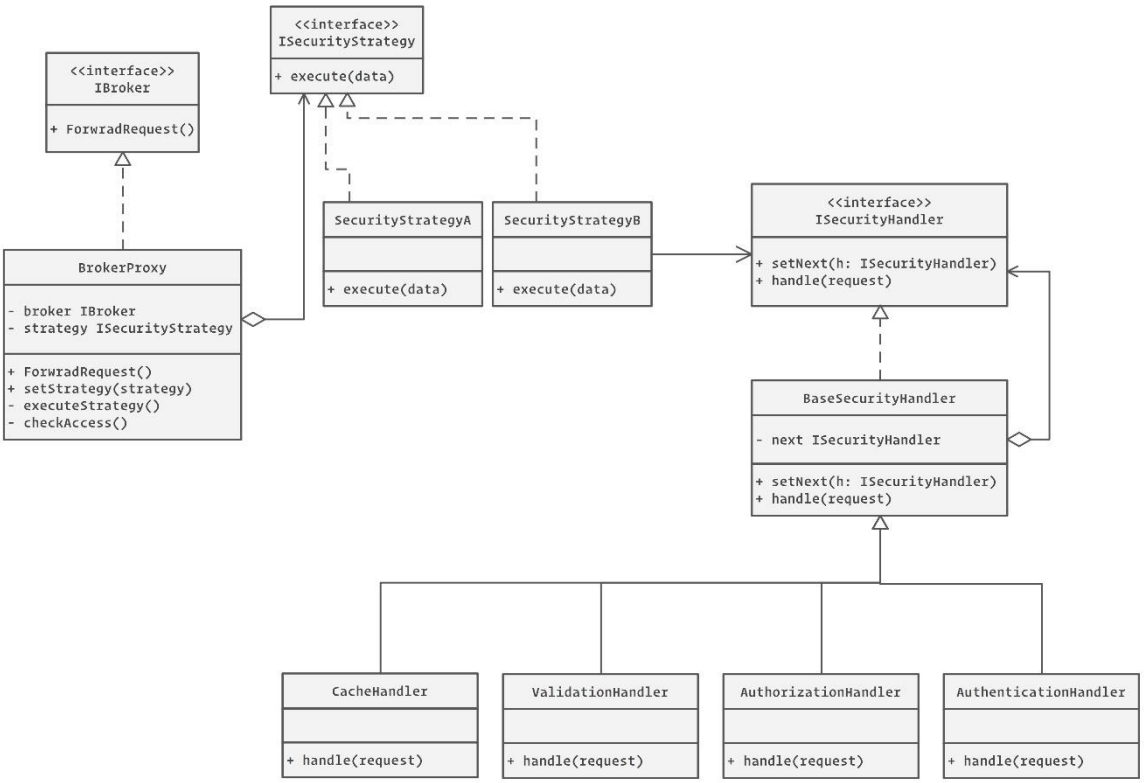


شکل ۲- نمودار کلاس سطح بالای مربوط به راه حل

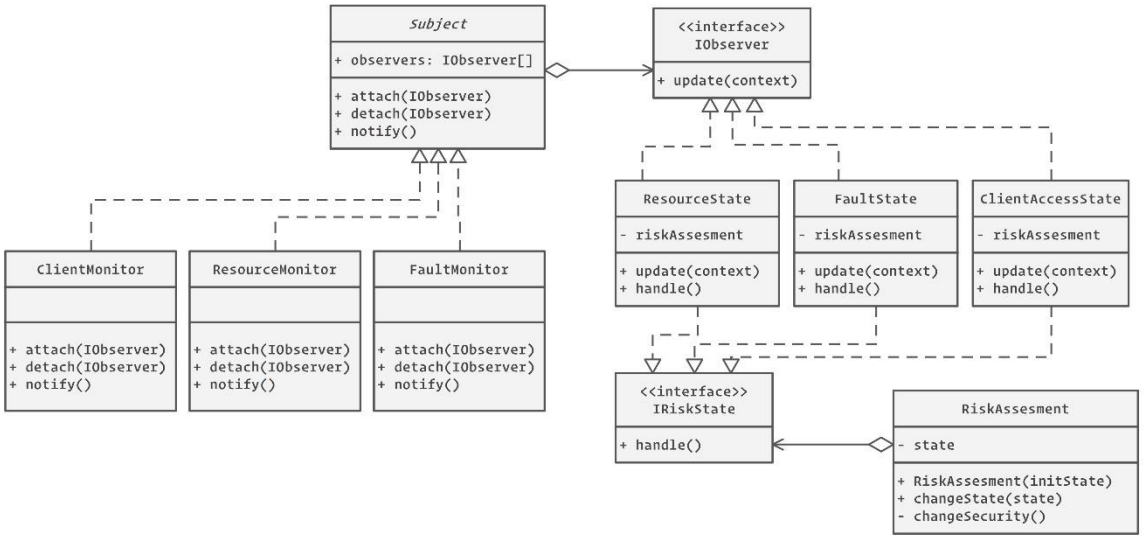
در شکل ۳ نمودار کلاس مربوط به نحوه اعمال استراتژی های مختلف مشخص شده است. Strategy ها تعیین کننده استراتژی های امنیتی و Handler ها کلاس های امنیتی ریزدانه برای ساخت سطوح امنیتی پیچیده هستند.

در شکل ٤ نمودار کلاس مربوط به نحوه اعمال اصول امنیتی مشخص شده است. در اینجا State ها تعیین کننده حالت هستند و اینها Observer برای Monitor های می باشند. RiskAssessment با تغییر حالت سیستم در صورت صلاحدید می تواند آن را به سیستم اعلام کند تا براساس آن سیستم مرکزی بتواند یک استراتژی امنیتی جدید براساس شرایط موجود در سیستم روی کل ساختار اعمال کند.

نمودار های کلاس به دلیل اینکه بتوانند در صفحات نشان داده شوند به این صورت شکسته شده اند و همه به طور کلی یک نمودار جامع هستند.



شکل ۳- نمودار کلاس نحوه اعمال استراتژی امنیتی



شکل ٤- نمودار کلاس تعیین استراتژی براساس مخاطره

مؤلفه های معماری و طراحی:

Server:

زیرسیستم مشترک است که به کلاینت ها سرویس می دهد. این زیر سیستم می تواند خود و سرویس هایش را در Broker ثبت کند، راه اندازی شود، وارد چرخه پاسخ دهی به درخواست ها گردد و درخواست های رسیده را انجام دهد و پاسخ مناسب را برگرداند. این سرور باید اینترفیس IService را پیاده سازی کند و کلاینت این اینترفیس را می شناسد. اینها با Server-Side Proxy در ارتباط هستند.

Client:

زیر سیستم هایی هستند که از سرور ها درخواست سرویس می کنند. کلاینت ها درخواست های خود را برای اجرا روی سرور ها به Broker ارسال کرده و اینترفیس Iservice را برای درخواست های خود می شناسند. اینها با Client-Side Proxy در ارتباط هستند.

Iservice:

اینترفیس سرویس هایی است که زیرسیستم مشترک می دهد را نشان می دهد و زیرسیستم های دیگر که کلاینت ها هستند این را می شناسند. چون در مسئله عنوان شده بود که یک زیرسیستم مشترک داریم یک اینترفیس برای آن تعریف شده ولی در صورتی چندین زیرسیستم مشترک یا سرور داشته باشیم به دلیل حفظ cohesive بودن اینترفیس باید تغییراتی در آن ایجاد گردد.

IBroker:

اینترفیس مشترک بین Broker و BrokerProxy است و هر دو باید عملیات آن را پیاده سازی کنند تا BrokerProxy بتواند کار را به Broker مربوط به خود delegate نماید.

Client-Side Proxy:

پروکسی سمت کلاینت است و برای آن نقش سرور را دارد و باعث ایجاد شفافیت می شود و کلاینت پروکسی را محلی می بیند.

این پروکسی باید اینترفیس IService را پیاده سازی کند تا کلاینت به آن درخواست ارسال نماید. همچنین این پروکسی پاسخ دریافتی را به کلاینت ارسال می کند. درخواست های کلاینت را به Broker ارسال و داده ها را بسته بندی و بازگشایی می کند. این پروکسی با Broker و BrokerProxy و Client در ارتباط است.

Server-Side Proxy:

پروکسی سمت سرور است و برای آن نقش کلاینت را دارد و باعث ایجاد شفافیت می شود و سرور کلاینت را محلی می بیند. این پروکسی درخواست دریافتی را به سرور ارسال و پاسخ دریافتی را به Broker باز می گرداند همچنین داده ها را بسته بندی و بازگشایی می کند. این پروکسی با Broker و Server در ارتباط است.

Broker:

Broker مسئول مدیریت تعاملات و ارتباطات بین کلاینت ها و سرور است. همچنین برای اینکه بتواند با BrokerProxy کار کند باید اینترفیس IBroker را پیاده سازی کند. در اینجا سرویس ها می توانند ثبت و حذف شوند و مخزن درخواست ها بروزرسانی شود و پاسخ ثبت و حذف سرویس به سرور ارسال شود. کلاینت و سرور را می تواند پیدا کند. درخواست ها و پاسخ ها را ارسال نماید. در نهایت Broker وارد حلقه پاسخگویی به درخواست ها می شود.

BrokerProxy:

مسئول تامین امنیت و اعمال سطوح امنیتی روی سیستم است. باید اینترفیس IBroker را پیاده سازی کند تا با Broker کار کند و درخواست ها از سمت کلاینت ابتدا به آن می رسد. و در صورتی توانست از زنجیره امنیتی رد شود درخواست به Broker داده می شود در غیر این صورت بدون درگیر کردن Broker پاسخ به کلاینت باز گردانده می شود. در پروکسی استراتژی های امنیتی از جانب مدیر سیستم یا براساس مخاطره تعیین می گردد و اجرا می شود.

ISecurityStrategy:

اینترفیس استراتژی های concrete را تعیین می کند و همه باید این را پیاده سازی کنند. BrokerProxy هم این اینترفیس را می شناسد.

SecurityStrategyX:

استراتژی های concrete هستند که در خود، الگوریتم های مختلف برای سطوح مختلف امنیتی را دارند و در شرایط مختلف سیستم یکی از اینها بسته به شرایط اجرا خواهد شد.

ISecurityHandler:

اینترفیس زنجیره امنیتی را تعیین می کند و عناصر زنجیره باید آن را پیاده سازی کنند که عملیات آن شامل اداره درخواست و تعیین عضو بعدی زنجیره است.

XHandler:

عناصر ریزدانه concrete زنجیره هستند که اینترفیس را پیاده سازی کرده اند و با ترکیب با هم زنجیره ای را برای ساخت الگوریتم برای سطوح مختلف امنیتی تشکیل می دهند. با رسیدن درخواست در صورتی که مورد امنیتی مورد نظرشان تامین شد درخواست را به عنصر بعدی زنجیره می دهند در غیر این صورت پاسخ مناسب را بر می گردانند که نشان از عدم رعایت شرایط امنیتی دارد و برای گذر و رفتن به Broker باید کل زنجیره طی و هر کدام درخواست مورد نظر را اجرا نمایند.

IObserver:

اینترفیس Observer را تعیین می کند و Observer ها باید آن را پیاده سازی کنند . subject هم این اینترفیس را می شناسد.

XMonitor:

اینها monitor هایی هستن که می توانند شرایط مختلف سیستم را در خود داشته باشد و Observer های concrete خود را برای دریافت تغییرات پیش اینها ثبت و در صورت تغییر حالت شان به آنها خبر داده می شود.

IRiskState:

اینترفیس state های مربوط به ریسک ها را پیاده سازی می کند. RiskAssessment این اینترفیس را می شناسد.

XState:

اینها state های concrete هستند که حالت های مختلف را براساس سطوح مخاطره تعیین می کنند. همچنین Observer برای monitor ها هستند و در صورت تغییرات در صورت لزوم حالت سیستم را تغییر می دهند.

RiskAssessment:

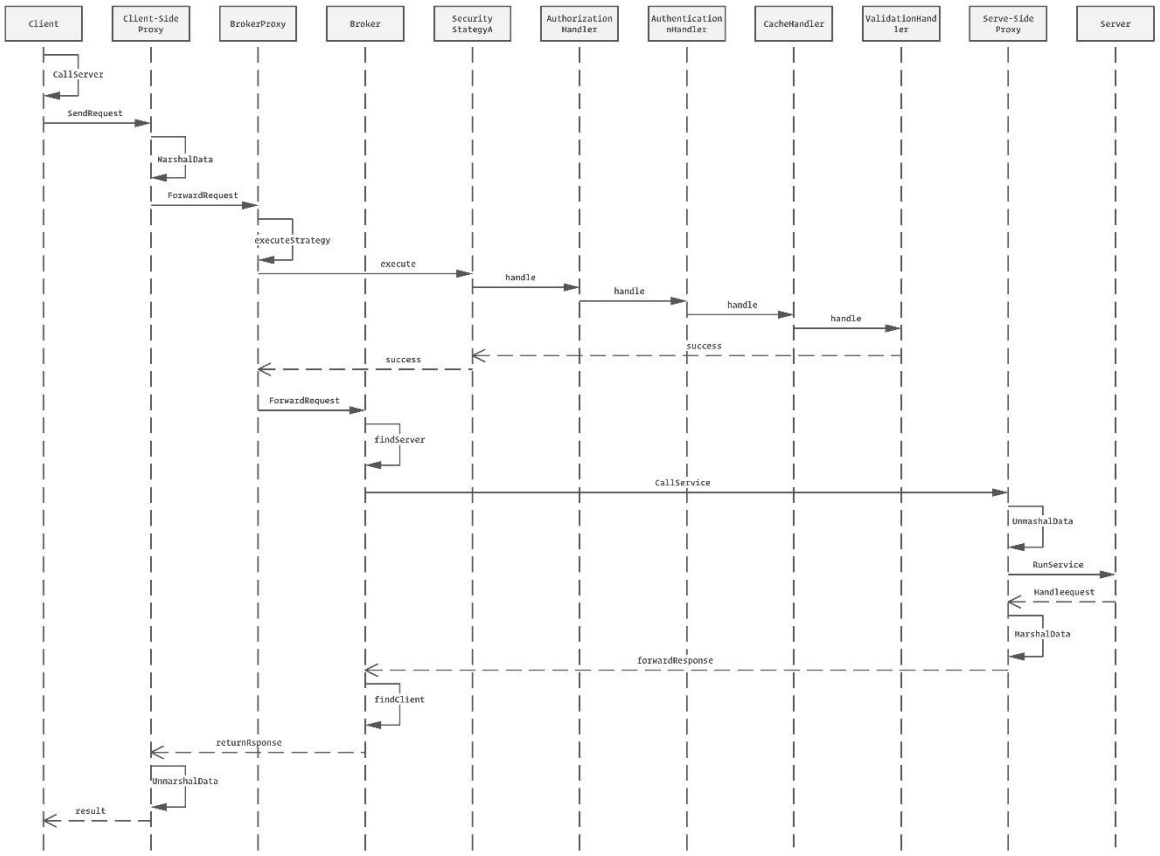
مسئول تعیین سطوح مخاطره در سیستم است و در صورت تغییر حالت متوجه تغییر حالت در سیستم می شود و در صورتی این تغییر حالت نیازمند تغییر در سطح مخاطره در سیستم باشد آن را اعلام میکند تا سیستم بتواند براساس آن استراتژی را به استراتژِی مناسب با این حالت تغییر بدهد.

سناریوها:

سناریو ۱: زیرسیستم درخواست را به زیرسیستم مشترک ارسال و پاسخ انجام سرویس را دریافت می کند

در شکل ۵ نمودار توالی مربوط به ارسال درخواست و دریافت پاسخ را می بینیم. در اینجا کلاینت ابتدا با سرور که اینجا برایش پروکسی سمت کلاینت است ارتباط برقرار می کند و درخواست را برای آن ارسال می کند. پروکسی درخواست را marshal کرده و آن را به Broker می فرستند. اینجا BrokerProxy که درخواست را دریافت می کند برایش نقش Broker را دارد. در اینجا پروکسی قبل از ارسال درخواست به Broker ابتدا استراتژی امنیتی مورد نظر را که برایش تعیین شده اجرا کرده و درخواست را به آن می دهد. در آنجا درخواست وارد زنجیره ای از بررسی های امنیتی ریزدانه می شود و درخواست تا انتهای زنجیره اجرا و همه را با موفقیت رد می کند. در انتهای زنجیره پیغام موفقیت برای BrokerProxy بازگردانده می شود. در اینجا BrokerProxy می تواند درخواست را برای اجرا به Broker بدهد.

بعد از دریافت درخواست در Broker ابتدا سرور مورد نظر که باید درخواست را پیدا کند مکانش پیدا می شود و بعد از آن درخواست برای اجرا به سمت آن هدایت می گردد. در آنجا درخواست در پروکسی سمت سرور دریافت می گردد. درخواست unmarshal شده و در خواست برای اجرا به سرور داده می شود. سرور درخواست را اجرا و پاسخ را به پروکسی بر می گرداند. در آنجا پاسخ marshal شده و پاسخ به Broker داده می شود. در Broker ابتدا کلاینت مکان یابی شده و پاسخ برایش ارسال می شود. پاسخ توسط پروکسی سمت کلاینت دریافت می شود unmarshal شده و نتیجه به کلاینت داده می شود.

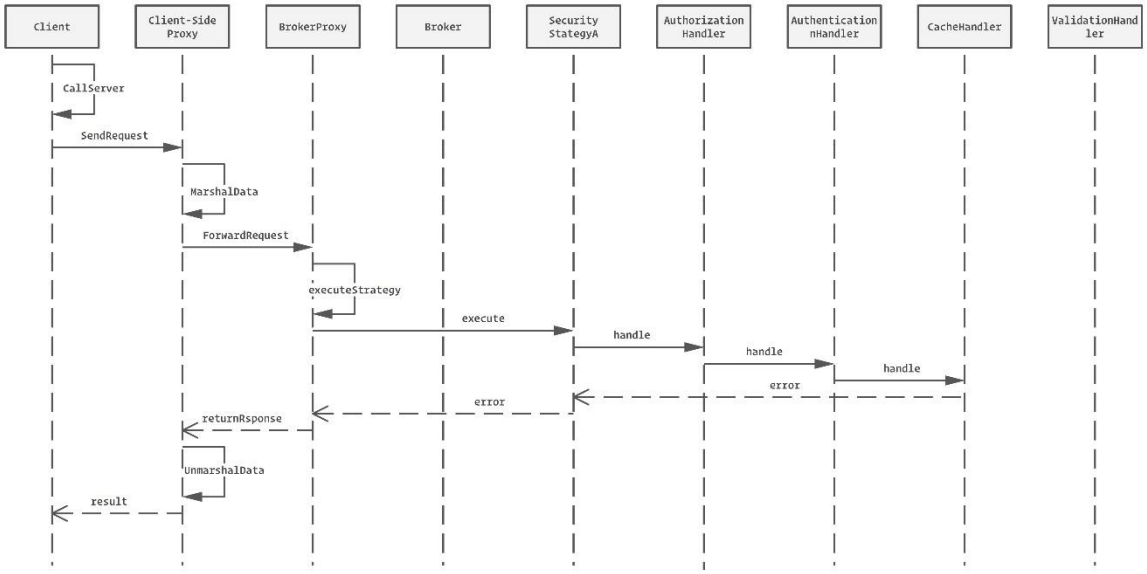


شکل ۵- نمودار توالی ارسال درخواست و دریافت پاسخ

سناریو ۲: زیرسیستم درخواست را به زیرسیستم مشترک ارسال اما به دلیل مشکل امنیتی پاسخ خطا دریافت می کند

در شکل ٦ نمودار توالی این سناریو را می بینیم. این سناریو تا بخشی مشابه سناریو ۱ است ولی وقتی وارد زنجیره بررسی های امنیتی می شود نمی تواند شرایط یکی از عناصر زنجیره را برآورده کند.

برای همین پاسخ مناسب درخواست به پروکسی بازگشت داده می شود و بدون اینکه درخواستی وارد پروکسی شود پیغام مناسب به کلاینت داده می شود.



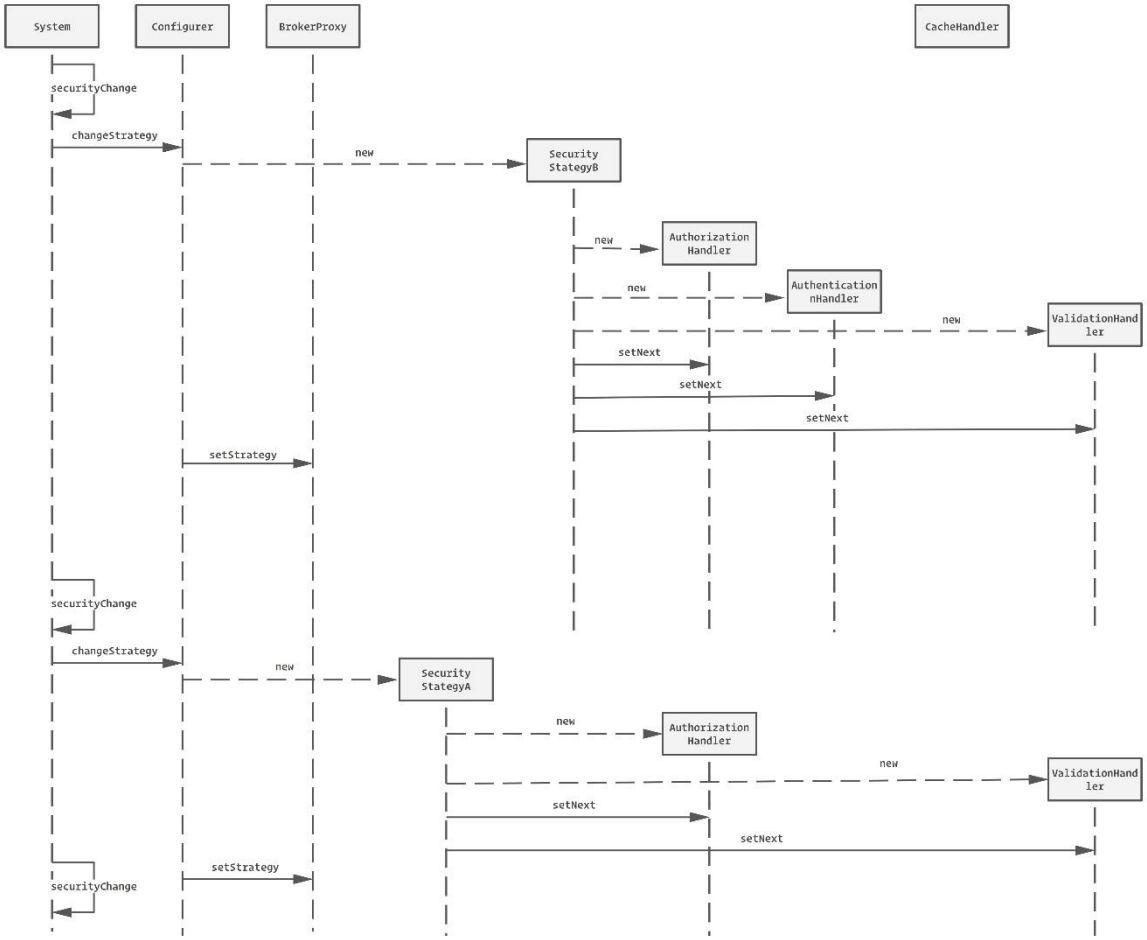
شکل ٦- نمودار توالی ارسال درخواست و عدم دریافت پاسخ

سناریو ۳: براساس تغییر شرایط امنیتی سطح جدید امنیتی تعیین می شود

در شکل ۷ سناریو مربوط به تعیین سطوح امنیتی را می بینیم. در اینجا سیستم درخواستی را برای تغییر سطح امنیت و بررسی آن به پیکربند می فرستد. پیکربند یک نمونه از استراتژی مورد نظر را می سازد. در خود استراتژی مورد نظر هم باید زنجیره بررسی های امنیتی از عناصر ریزدانه ساخته شود. در استراتژی ابتدا عناصر ریز دانه ساخته شده و سپس ترتیب زنجیره مشخص می گردد. بعد از پایان ساخت استراتژِی ، شی ساخته شده به BrokerProxy داده می شود تا بتواند آن روی درخواست ها اعمال نماید. در ادامه می بینیم که بعد از مدتی درخواست جدیدی برای تعیین استراتژی جدید از سیستم به پیکربند داده می شود و پیکربند هم استراتژی جدید را ساخته و آن را برای اعمال به BrokerProxy می دهد.

سناریو ٤: براساس تغییر در حالت سیستم سطح مخاطره تغییر می کند

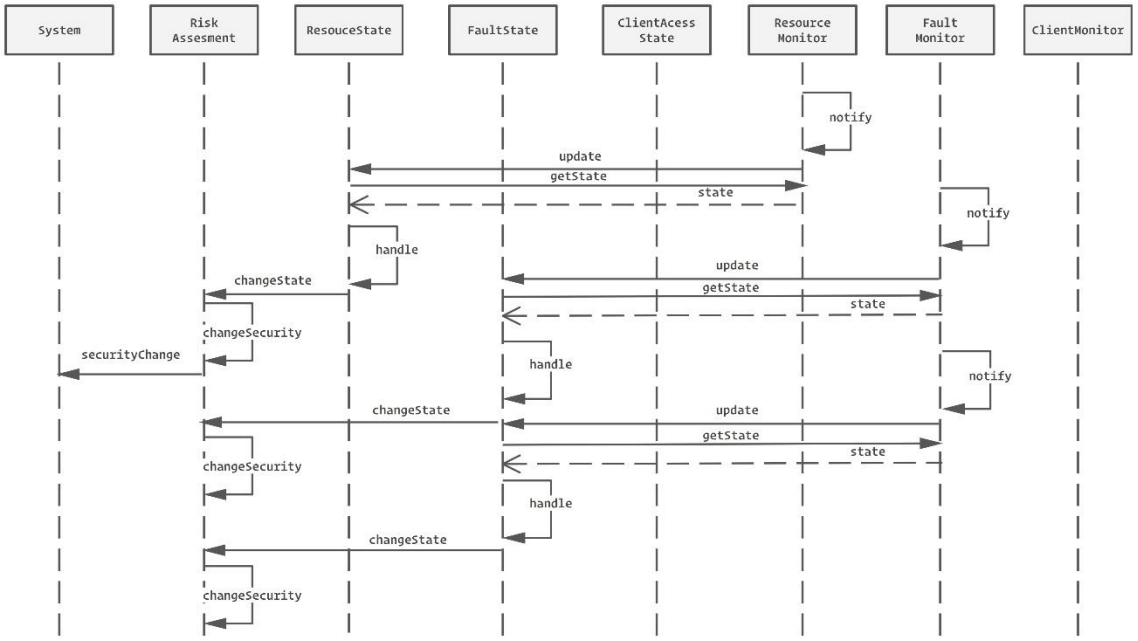
در شکل ۸ نمودار توالی مربوط به تعیین سطح مخاطره در سیستم می بینیم. در اینجا State ها که Observer برای Monitor ها هستند از قبل پیکربندی شده اند و منتظر دریافت تغییرات آنها برای تغییر حالت سیستم هستند.



شکل ۷- نمودار توالی تعیین سطوح امنیتی

در یکی از monitor ها تغییری رخ می دهد و آن به تمام Observer ها پیغام update می دهد. Observer مورد نظر پیغام update را دریافت و حالت monitor را می گیرد. این observer که state هم هست شرایط را بررسی و طبق آن درخواست تغییر سطح مخاطره را به سیستم اعلام می کند تا براساس آن تصمیم بگیرد.

در ادامه تغییر در یکی دیگر از monitor ها رخ می دهد و همین کار برای این هم تکرار می شود اما RiskAssessment تصمیم می گیرد تغییر در سطح مخاطره ایجاد نشود. در ادامه تغییری در monitor که ابتدا باعث تغییر سطح مخاطره می شود رخ می دهد اما در اینجا هم تصمیم گرفته می شود سطح مخاطره تغییر نکند و بنابراین چیزی به سیستم اعلام نمی شود.



شکل ۸- نمودار توالی تعیین سطوح مخاطره

معایب راه حل:

۱. ***کاهش کارایی:*** کارایی نسبت به حالت قبل که مستقیم با هم ارتباط برقرار می کردند کاهش می یابد. به دلیل اینکه بیشتر جاها غیر مستقیم شده و موارد زیاد اشیا جعلی بین کلاینت و سرور قرار گرفته اند کارایی کاهش یافته اما این باعث شده موارد بسیار زیادی عاید سیستم شود. البته در همین راه حل امکان تبدیل شدن BrokerProxy و Broker به Single Point of Failure و Bottleneck شدن وجود دارد. البته می توان با کمک replication مشکل را حل کرد ولی چون بعضی موارد مربوط به بررسی های امنیتی باید حتما مرکزیت کامل داشته باشند ممکن است نتوان در تمام شرایط replication را به کار برد.

۲. ***تعداد اشیا زیاد:*** تعداد اشیا نسبت به حالت قبل بسیار زیاد تر شده است. به علاوه تعداد زیادی اشیا هم در طول اجرا ممکن است ساخته شود که این مورد هم می تواند حافظه مصرفی و هم کارایی را کاهش بدهد. یک راه حل می تواند استفاده از الگوی Prototype باشد و اشیا و پیکربندی های متنوع و ممکن به خصوص در مورد استراتژی ها از قبل ساخته شود و در مواقع لزوم از آنها کپی گرفته شود که می تواند در افزایش کارایی بسیار مفید باشد.

۳. ***نیاز به پیکربندی:*** در الگوریتم نیاز به پیکربندی وجود دارد و در حالت قبل نیاز به این کار نبود. البته این کار انعطاف پذیری را بسیار بالا می برد اما به هر حال مسئولیت جدید است که به سیستم اضافه شده و بدون پیکربندی اولیه و در حال اجرا امکان استفاده از راه حل نیست و همچنین خود پیکربندی هم می تواند دشوار باشد.

۴. ***پیام رسانی زیاد:*** در راه حل به دلیل زیاد شدن تعداد اشیا و به خصوص غیر مستقیم شدن ارتباطات تعداد پیام ای ارسالی آنها زیاد تر شده است. به خصوص در بخش تعیین سطح مخاطرات که نیازمند آگاهی از تغییرات برخی موارد در سیستم است این مورد می تواند بسیار بیشتر باشد. همچنین در صورتی که زنجیره های بررسی امنیتی هم بزرگ شوند تعداد پیام های ارسالی میان اشیا بسیار زیاد خواهد بود.

۵. ***وجود عنصر مرکزی:*** وجود عنصر مرکزی Broker که تمام تعاملات را برعهده دارد علاوه بر مزایای فراوانی که ایجاد می کند می تواند در صورتی خرابی باعث شود کل سیستم از کار بیفتد و تا جایگزین کردن آن هم هیچ راهی برای ارتباط میان کلاینت ها و سرور وجود نخواهد داشت و به طور کل می توان گفت تحمل پذیری خطا در سیستم پایین است.

۶. ***کاهش قابلیت آزمون:*** به دلیل ریزدانه شدن بخش های مختلف و ارتباطات زیاد و برخی الگو های استفاده شده به طور کلی قابلیت آزمون و عیب یابی نسبت به قبل کاهش می یابد و نیازمند صرف انرژی و وقت بیشتر برای این کار است.

مزایای راه حل:

۱. ***کاهش Coupling:*** به دلیل اینکه وابستگی در بیشتر موارد به اینترفیس ها است در راه حل وابستگی کمی وجود دارد. به خصوص نسبت به حالت قبل وابستگی های زیرسیستم ها و زیرسیستم مشترک کاملا از بین رفته است و به طور کلی وابستگی در راه حل بسیار پایین است.

۲. ***افزایش cohesion:*** در راه حل بیشتر مولفه ها تک کاره و تک منظوره هستند که این نشان دهنده بالا بودن cohesion در سیستم است.

۳. ***کاهش پیچیدگی ها:*** ریز دانه شدن و جداسازی بخش های مختلف پیچیدگی ها را به اندازه زیادی کاهش داده است. البته با زیاد شدن تعداد کلاس ها ممکن است این طور به نظر برسد که پیچیدگی افزایش یافته اما به دلیل بالا بودن cohesion وجود ریزدانگی ها باعث نشده پیچیدگی افزایش یابد و پیچیدگی ها کاهش یافته است.

۴. ***افزایش قابلیت آزمون:*** به دلیل ریزدانه شدن بخش های مختلف و cohesive بودن آنها قابلیت آزمون در سطح ریز دانه افزایش یافته است اگرچه در سطح کل سیستم این مورد کاهش داشته است.

۵. ***قابلیت پیکربندی:*** این مورد جز معایب هم بود اما خود وجود پیکربندی می تواند مزیت باشد و می توان سیستم را به حالات مختلفی پیکربندی کرد که برای سیستم بسیار مفید است.

۶. ***انعطاف پذیری:*** قابلیت پیکربندی در زمان اجرا می تواند کمک کند بدون نیاز به کامپایل مجدد کد شرایط مناسب سیستم را بدست آورد و از این لحاظ مزیت بزرگی است.

۷. ***جدایی دغدغه ها:*** تمام بخش های ریز دانه دغدغه های خود را دارند و دغدغه بخش های مختلف از هم جدا شده که این مورد باعث cohesive شدن بخش های مختلف سیستم شده است.

۸. ***افزایش indirection:*** این مورد می تواند باعث کاهش کارایی شود اما به دلیل غیر مستقیم شدن ها و وابستگی ها با اینترفیس ها این کار باعث افزایش تغییر پذیری بخش های مختلف در سیستم می شود.

۹. ***افزایش تغییر پذیری:*** به دلیل غیر مستقیم شدن روابط، cohesive بودن و جدا بودن دغدغه ها و عدم وابستگی ها به پیاده سازی تغییرات بسیار ساده تر شده است و انجام تغییرات کم ترین میزان انتشار تغییر را خواهد داشت.

۱۰. ***قابلیت استفاده مجدد:*** کاهش وابستگی ها و افزایش انسجام و عدم انتشار تغییرات و وابستگی ها به اینترفیس باعث می شود موارد زیادی در راه حل پیشنهاد قابلیت استفاده مجدد داشته باشند.

معرفی دو idiom در C++:

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 خاص هستند و می تواند پیاده سازی concrete یک الگوی طراحی باشد و اینها راه حل های درست مشکلات سطح پایین در ارتباط با یک زبان خاص هستند.

در ادامه دو idiom با نام های virtual constructor و Parameterized Base Class مربوط به زبان برنامه سازی C++ معرفی خواهد.

Virtual Constructor:

(این idiom به نام ***Factory Method of initialization*** نیز شناخته شده است)

هدف:

ساخت کپی از یک شی یا شی جدید بدون دانستن نوع concrete آن است.

انگیزه:

کاربرد فراخوانی توابع در ساختار سلسله مراتبی کلاس ها به صورت چند ریختی در شی گرایی به خوبی شناخته شده است.

این روش پیاده سازی رابطه is-a یا به طور عملی تر behaves-as-a است. گاهی فراخوانی توابع مدیریت چرخه حیات ساختار سلسله مراتبی کلاس ها مثل construction، destruction و copy به صورت چند ریختی بسیار مفید است.

c++ به طور بومی از destruction چند ریختی اشیا با استفاده از virtual destructor پشتیبانی می کند اما معادل آن برای ساخت و کپی اشیا را پشتیبانی نمی کند.

در c++ برای ساخت اشیا نیاز است که نوع آن در زمان کامپایل شناخته شود. idiom به نام Virtual Constructor اجازه ساخت و کپی اشیا را به صورت چند ریختی در c++ می دهد.

راه حل:

اثر یک Virtual Constructor توسط تابع ()create برای ساخت و تابع ()clone برای کپی در کد زیر نشان داده شده است.

```
class Employee
{
  public:
    virtual ~Employee () {} // Native support for polymorphic destruction.
    virtual Employee * create () const = 0; // Virtual constructor (creation)
    virtual Employee * clone () const = 0;  // Virtual constructor (copying)
};
class Manager : public Employee     // "is-a" relationship
{
  public:
    Manager ();                     // Default constructor
    Manager (Manager const &);      // Copy constructor
    virtual ~Manager () {}          // Destructor
    Manager * create () const       // Virtual constructor (creation)
    {
      return new Manager();
    }
    Manager * clone () const        // Virtual constructor (copying)
    {
      return new Manager (*this);
    }
};
class Programmer : public Employee { /* Very similar to the Manager class */ };
Employee * duplicate (Employee const & e)
{
   return e.clone();  // Using virtual constructor idiom.
}
```

کلاس Manager دو تابع مجازی خالص را پیاده سازی می کند و از نام نوع (Manager) برای ساخت آنها استفاده می کند. تابع تکراری نحوه استفاده از idiom را نشان می دهد اما در واقع نمی داند چه چیزی را تکرار می کند و تنها می داند یک Employee را تکثیر می کند. وظیفه ساخت کلاس درست به نمونه مشتق شده از آن واسپاری شده است. بنابراین تابع تکراری برای تغییرات بسته است حتی اگر ریشه ساختار سلسله مراتبی کلاس که Employee است در آینده زیرکلاس های بیشتری را اضافه کند.

نوع بازگشتی از توابع ساخت و تکثیر کلاس Manager از نوع Employee نیست بلکه از نوع خود کلاس است. c++ اجازه این انعطاف پذیری در نوع را در جایی که نوع بازگشت تابع override شده می تواند یک نوع مشتق شده از تابع در کلاس پایه باشد را می دهد. این ویژگی زبان به نام co-variant return types شناخته می شود.

برای اداره صحیح مالکیت منبع باید idiom به نام Resource Return برای نوع بازگشت توابع clone() و create() به کار گرفته شود زیرا اینها توابع Factory هستند. اگر Resource Return استفاده شود نوع های بازگشت یعنی (shared_ptr<Employee> و shared_ptr<Manager>) دیگر نوع های بازگشت هم گونه نیستند و برنامه باید برای کامپایل دچار خطا بشود. در این گونه موارد تابع virtual constructor در کلاس مشتق شده باید دقیقا نوع کلاس پدر را برگرداند.

```
#include <tr1/memory>

class Employee
{
  public:
    typedef std::tr1::shared_ptr<Employee> Ptr;
    virtual ~Employee () {}   // Native support for polymorphic destruction.
    virtual Ptr create () const = 0; // Virtual constructor (creation)
    virtual Ptr clone () const = 0;  // Virtual constructor (copying)
};
class Manager : public Employee     // "is-a" relationship
{
  public:
    Manager () {}                     // Default constructor
    Manager (Manager const &) {}      // Copy constructor
    virtual ~Manager () {}
    Ptr create () const       // Virtual constructor (creation)
    {
      return Ptr(new Manager());
    }
    Ptr clone () const        // Virtual constructor (copying)
    {
      return Ptr(new Manager (*this));
    }
};
```

تعریف اصطلاحات:

تابع مجازی: عنصری کلیدی در پارادایم شی گرایی است که فراخوانی کد قدیمی توسط کد جدید را ساده می کند. توابع مجازی به کلاس های مشتق شده اجازه می دهد کد پیاده سازی شده در کلاس پایه را جایگزین کنند.

تابع مجازی خالص: تابعی است که باید در کلاس مشتق شده override شود و نیاز نیست تعریف شود. یک تابع مجازی با استفاده از 0= به یک تابع مجازی خالص تبدیل می شود.

Parameterized Base Class:

(این idiom به نام ***Mixin-from-below*** و ***Parameterized Inheritance*** نیز شناخته شده است)

هدف:

Abstract کردن یک وجه در یک پیمانه قابل استفاده مجدد و ترکیب آن با نوع داده شده در زمان لزوم

انگیزه:

یک وجه خاص از نیازمندی ها را می توان انتزاع کرد و آن را به شکل قالب ایجاد کرد (مانند object serialization). Serialization یک دغدغه چند جانبه است که خیلی از کلاس ها یا انواع POD در اپلیکیشن ها می توانند داشته باشند. چنین دغدغه هایی می توانند در یک پیمانه قابل استفاده مجدد قابل مدیریت انتزاع شوند. با اضافه شدن یک وجه، قابلیت جایگزینی با نوع اصلی از بین نمی رود بنابراین انگیزه دیگر این است که یک رابطه توارث عمومی (is-a) یا توارث خصوصی (was-a) داشته باشیم.

راه حل:

```
template <class T>
class Serializable : public T,   /// Parameterized Base Class Idiom
                     public ISerializable
{
  public:
    Serializable (const T &t = T()) : T(t) {}
    virtual int serialize (char *& buffer, size_t & buf_size) const
    {
        const size_t size = sizeof (T);
        if (size > buf_size)
          throw std::runtime_error("Insufficient memory!");

        memcpy (buffer, static_cast<const T *>(this), size);
        buffer += size;
        buf_size -= size;
        return size;
    }
};
```

در اینجا Serializable<T> می تواند به طور چند ریختی به عنوان T و ISerializable استفاده شود. مثال بالا به درستی کار می کند تنها در صورتی که T یک نوع POD بدون اشاره گر تعریف شده توسط کاربر باشد.

تعریف اصطلاحات:

POD: مخفف Plain Old Data کلاسی است که با کلمه کلیدی class یا struct تعریف می شود اما constructor، destructor و تابع مجازی ندارد. یک نوع POD اگر تنها شامل انواع built-in یا ترکیب آنها باشد.

منابع:

1. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley, 1995.

2. Alexander Shvets, Dive Into Design Patterns, 2019

3.Hans Rohnert, Michael Stal, Regine Meunier, Peter Sommerlad, Frank Buschmann, Pattern Oriented Software Architecture Volume 1: A System of Patterns Volume 1 Edition , Wiley, 1996

4. https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Virtual_Constructor