الگوها در مهندسی نرم‌افزار

تمرین دوم

استاد:

دکتر رامسین

دانشجو:

عماد خالق پناه

۹۸۲۰۹۴۵۶

فهرست

سوال اول: مسئله‌ی طراحی معماری با ترکیب الگوها

مسئله

در یک سیستم توزیع شده، زیرسیستم‌ها به طور پیچیده و کنترل نشده‌ای با یکدیگر در تعامل هستند. می‌خواهیم سیستم طوری طراحی شود که بتوان به صورت پویا دستیابی به بعضی از سرویس‌های هر زیرسیستم را برای زیرمجموعه‌ی مشخصی از مشتریانش محدود کرد. این محدودیت‌ها باید در زمان اجرا به صورت مستمر و براساس وضعیت سیستم از نظر توزیع بار کاری و سابقه دستیابی‌های مشکل‌زا بین زیرسیستم‌ها، مثل درخواست‌های خطا، مورد بازنگری قرار گرفته و اِعمال شوند.

زیرمسئله‌ها و راه حل پیشنهادی برای هر بخش

- **توزیع شدگی**: برای مدیریت توزیع شدگی از الگوی broker استفاده می‌کنیم. می‌توان فرض کرد روی هر گره از شبکه یک یا چند زیرسیستم قرار دارند. برای ارتباط با زیرسیستم‌های روی همان ماشین از طریق local broker و برای ارتباط با دیگر گره‌های شبکه از bridgeها در دو طرف ارتباط، استفاده می‌شود. هر زیرسیستم در داخل خود تعدادی server دارد که سرویس ارائه می‌کنند و تعدادی client که از دیگر serverهای این زیرسیستم یا دیگران درخواست سرویس می‌کنند. بنابراین از مزایای این الگو که پنهان بودن مکان فیزیکی زیرسیستم‌ها و سرورها از دید یکدیگر است استفاده می‌کنیم. همچنین از قابلیت تعویض و گسترش اجزا، جابجایی پذیری اجزا، قابلیت استفاده مجدد و ... بهره می‌بریم. همچنین برای داشتن cohesion بالا فرض می‌کنیم که هر server یک سرویس خاص ارائه می‌کند یا چند سرویس مشابه که از داده های یکسان یا همبسته استفاده می‌کنند. در قسمت کنترل دسترسی از این موضوع استفاده می‌کنیم.

- **ارتباطات پیچیده و کنترل نشده:** برای مدیریت پیچیدگی از الگوی mediator کمک می‌گیریم. به این صورت که bridgeهای هر گره شبکه، نیاز به ارتباط مستقیم با هم و اطلاع از آدرس فیزیکی یکدیگر ندارند. در واقع وقتی که local broker توسط کلاینت فراخوانی می‌شود، اگر سرویس مربوطه در آن گره موجود نباشد، درخواست را از طریق bridge که جزئیات شبکه‌ای را پیاده سازی کرده به mediator فرستاده می‌شود. عنصر mediator اطلاعات همه‌ی گره‌ها، زیرسیستم‌ها و سرویس‌هایی که ارائه می‌دهند را می‌شناسد و گره مربوطه را شناسایی و درخواست را به bridge مربوط به آن گره از شبکه می‌فرستد.

mediator خود نیز از ساختار الگوی broker پیروی می‌کند، یعنی دارای broker و bridge و server و ... برای ارتباط با زیرسیستم‌ها است. اما برای سادگی در توضیحات از تشریح ساختار broker و server و ... مربوط به آن خودداری کرده‌ایم.
همچنین روی زیرسیستم می‌توان یک facade تعریف کرد. اما به دلیل وجود proxyها و brokerها، برای بیشتر نشدن سطح indirection، از آن صرف نظر می‌کنیم. دلیل دیگر این کار این است که می‌توان هر گره شبکه را معادل یک زیرسیستم در نظر گرفت و عملا broker معادل facade روی زیرسیستم می‌شود. همچنین، به دلیل بار کاری زیادی که روی زیرسیستم‌ها وجود دارد، می‌توان فرض معادل بودن هر گره شبکه با یک زیرسیستم را اجباری در نظر گرفت و در مدل ارائه شده هر زیرسیستم مستقلا روی یک گره شبکه باشد. بنابراین در ادامه‌ی توضیحات و مدل‌ها و نمودارهای ارائه شده، این فرض در نظر گرفته شده است.

- **مشتری بیرونی:** مشتری های بیرونی نیز به عنوان یک زیرسیستم در نظر گرفته می‌شوند که فقط client دارند. آنها نیز فقط mediator را می‌شناسند و در واقع این mediator برای مشتری‌های بیرونی نقش یک gateway را بازی می‌کند. از این به بعد مشتری بیرونی را با زیرسیستمی با نام Customer مشخص می‌کنیم. مشتری می‌تواند برنامه را روی ماشین‌های خود که در یک گره دلخواه در شبکه است اجرا کند و سایر زیرسیستم‌ها و mediator مستقل از جزئیات سطح ماشین آن هستند و تحت عنوان زیرسیستم Customer آن را می‌شناسند که می‌تواند چندین نمونه در جاهای مختلف شبکه داشته باشد. در واقع ساختار کلی این سیستم اینگونه در نظر گرفته شده است که کلاینت از طریق mediator درخواست یک سرویس از سیستم می‌کند، و یک زیرسیستم به تنهایی یا به کمک سرویس‌گیری از زیرسیستم های دیگر به نیاز مشتری جواب می‌دهد. می‌توان به طور کامل مشتری را نیز از سیستم حذف کرد و اینگونه در نظر گرفت که این زیرسیستم‌ها برای اجرای یک هدف خاص در حال کار هستند و با هم در تعامل هستند و به این صورت نیست که یک کلاینت مستقیما به سیستم درخواست دهد و منتظر جواب از سیستم باشد.

- **کنترل دسترسی:** از proxy برای این کار استفاده می‌کنیم. می‌توان این کار را به اجزای موجود سپرد و از افزودن یک لایه‌ی indirection جدید که سربار را زیاد می‌کند خودداری نمود. راه حل پیشنهادی سپردن آن به server-side-proxy است. زیرا در این صورت می‌توان دسترسی را برای هر زیر مجموعه

ای از سرویس های یک زیرسیستم محدود کرد، نه فقط تمام آن (کاملا منطبق با صورت سوال). بنابراین server-side-proxy دسترسی به سرویس را برای هر درخواست بررسی کرده و بر اساس آن سرویس می‌دهد یا پیام دسترسی غیرمجاز را برمی‌گرداند. میتوان برای موارد حیاتی که کل زیرسیستم دچار مشکل شده است یا کلا گره شبکه و تمام زیرسیستم های آن مشکل دارند، در سطح bridge یا brokerها دسترسی را برای همه‌ی درخواست‌ها یا بخشی از درخواست‌ها بر اساس وضعیت سیستم قطع کرد. برای سادگی، در ادامه از تفصیل این حالت کلی که توسط bridge یا broker مدیریت می‌شود صرف نظر کرده و آن را بدیهی فرض می‌کنیم.

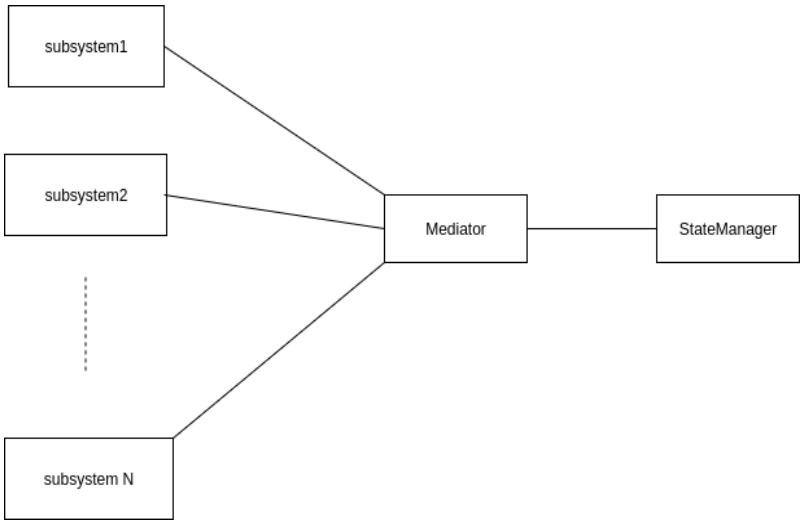
در صورت سوال به کنترل دسترسی برای زیرمجوعه‌ای مشخص از سرویس گیرنده‌ها اشاره شده است. از این جمله برداشت می‌شود که می‌توان یک لیست از اسامی زیرسیستم‌ها به عنوان blocklist به ازای هر سرویس یا سرور یا زیرسیستم تعریف کرد و به آنها نسبت داد و سیستم را طوری طراحی کرد که در هر درخواست نام سرویس گیرنده نیز ذکر شود تا در مقصد بتوان در صورت نیاز به قطع دسترسی، بودن یا نبودن آن سرویس گیرنده در لیست مذکور را چک کرد. ما در اینجا یک trade-off بین سپردن همه ی مسئولیت‌های ارتباطات به سطوح بالاتر مثل mediator و مدیریت ارتباطات توسط سطوح پایین‌تر که زیرسیستم‌ها یا سرورها و کلاینت‌های تشکیل دهنده آن هستند، داریم و برای جلوگیری از پیچیدگی بیش‌ازحد mediator و God Class شدن آن، برخی از مسئولیت‌ها مانند همین مسئله‌ی blocklist را در سطوح پایین‌تر قرار می‌دهیم.

- **نگه‌داری وضعیت سیستم و زیرسیستم‌ها و سابقه‌ی دسترسی‌های مشکل‌زا و کنترل دسترسی پویا بر اساس آنها:** وضعیت‌ها را با ترکیبی از state و strategy و observer مدیریت می‌کنیم. یک زیرسیستم تحت عنوان StateManager طراحی می‌کنیم که از طریق mediator به همه‌ی سرویس‌ها دسترسی دارد. SateManager نیز از ساختار الگوی broker پیروی می‌کند اما برای سادگی در توضیحات از تشریح ساختار الگوهای broker و server و ... مربوط به آن خودداری کرده‌ایم. برای هرserver-side-proxy به عنوان context، یک state در نظر می‌گیریم که فرزندانی به اسم NormalState و CriticalState دارد. زیرسیستم StateManager می‌تواند براساس مانیتور کردن پارامترهای کلی سیستم مانند توزیع بار کاری به زیرسیستم‌های مورد نظر پیام دهد که وضعیت برخی از serverهای خود را در حالت اضطراری قرار دهد (الگوی observer) و یا این کار با تشخیص خود زیرسیستم، مثلا به وسیله‌ی broker، از طریق ارسال پیام به server-side-proxy یا توسط server-

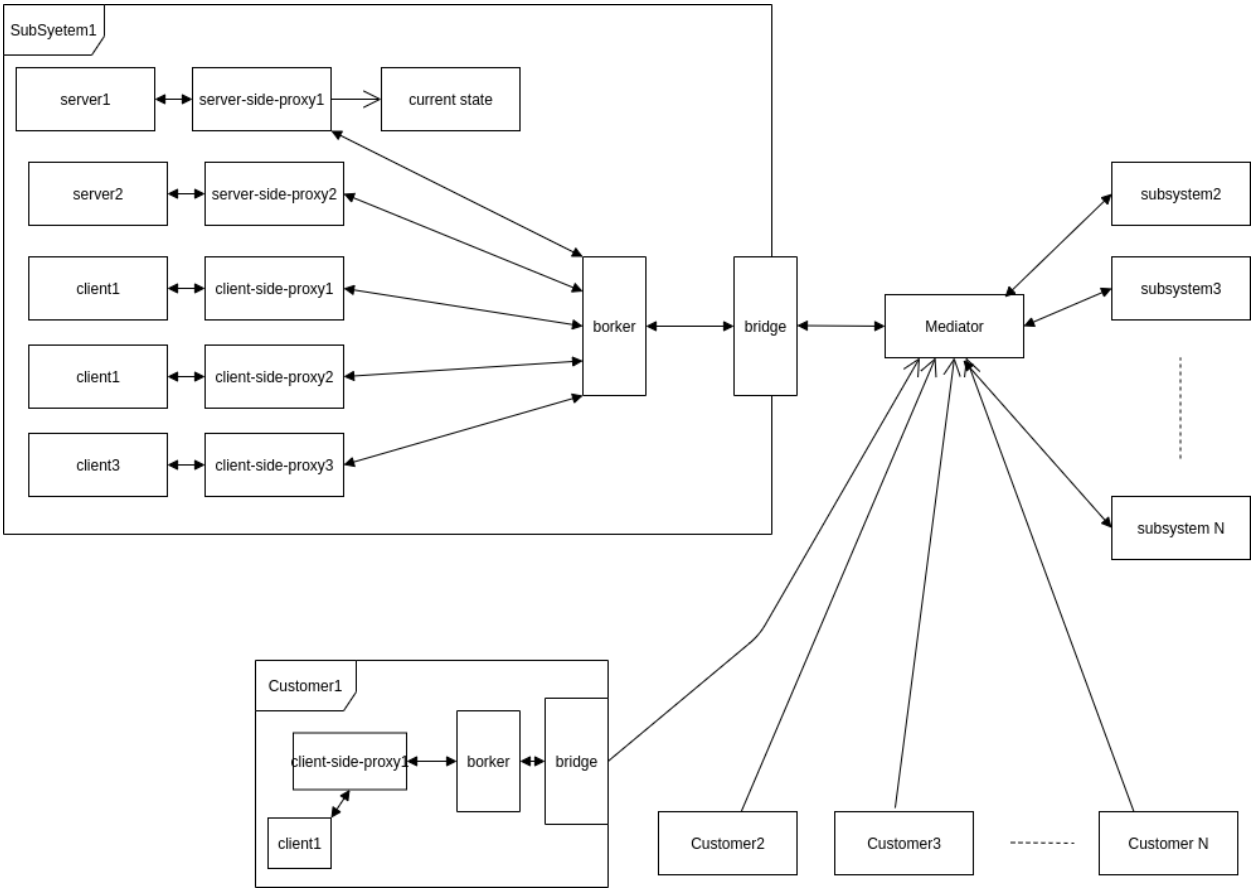
side-proxy، براساس داده‌های خودشان مانند سابقه‌ی دسترسی‌های مشکل‌زا و خطاها که در طول زمان و براساس درخواست‌های رد شده جمع آوری شده انجام شود. در حالت نرمال، server به همه‌ی درخواست‌ها جواب می‌دهد و در حالت اضطراری به سرویس‌گیرنده‌های موجود در blocklist خود سرویس نمی‌دهد. می‌توان این لیست را بر اساس سرویس‌ها نیز جدا کرد، یعنی هر سرویس یک لیست محدودیت جدا داشته باشد ولی به دلیل پیچیدگی زیاد این کار و همچنین cohesive بودن serverها (در انتهای زیرمسئله‌ی توزیع شدگی به این مورد پرداخته شد) کنترل دسترسی را بر اساس server در نظر می‌گیریم.

نمودارها و بررسی دقیقتر اجزا

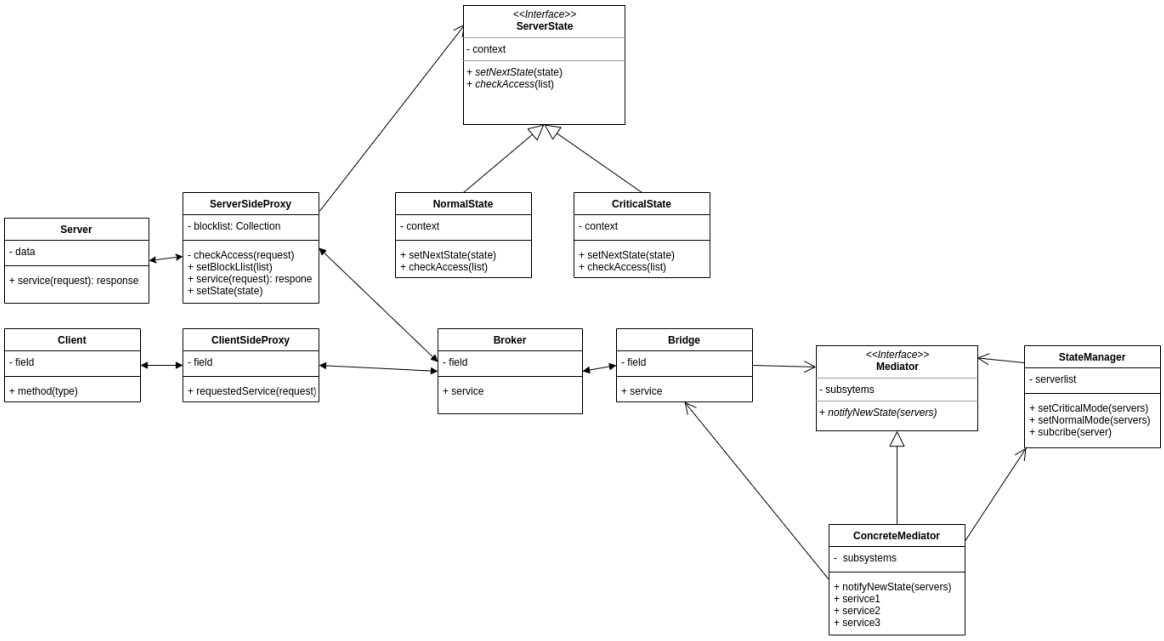
ساختار کلی زیرسیستم‌ها در شکل زیر قابل مشاهده است:



در شکل دوم، جزئیات درون زیرسیستم‌ها بر اساس الگوهای ذکر شده نشان‌داده‌است:



و در نهایت یک نمودار کلاسی به صورت کلی و بدون تعریف دقیق رفتارها و داده‌های کلاس‌ها و صرفا برای نشان دادن ارتباطات در شکل زیر مشاهده می‌شود. اجزای الگوی broker به دلیل ماهیت معماری داشتن، به خوبی در این شکل قابل مشاهده نیستند. به صورت دقیق‌تر همه‌ی این اجزا از server تا bridge، باید کلاس‌های گوناگونی بر اساس کارایی زیرسیستم مورد نظر داشته باشند که به کمک وراثت و استفاده از واسط و ... طراحی شوند. در این جا صرفا یک دید کلی به مسئله نشان داده شده است.



توضیحات کافی برای چگونگی ارتباط اجزا و ماهیت اجزا در شکل های بالا در قسمت زیرمسئله ها آورده شده است.

در ادامه چند سناریو را بررسی می‌کنیم. به دلیل تعداد زیاد لایه‌ها و دشوار بودن رسم یک sequence diagram خوانا و همچنین، عدم خوانا بودن نموداری که با حذف برخی لایه های کم اهمیت قابل رسم است، سناریوها را بدون شکل اما به صورت دقیق تشریح می‌کنیم.

سناریو اول: پیکربندی اولیه سیستم

در ابتدا brokerها اجرا شده و در event loop منتظر درخواست می‌مانند. باید کد serverها بعد از اجرا شدن خود را به broker معرفی کنند و او اطلاعات لازم را ذخیره کرده و به serverها Ack می‌فرستد. blocklist در serverها تعیین می‌شوند و State اولیه serverها در حالت نرمال set می‌شود. هر broker نیز اطلاعات زیرسیستم خود مانند لیست serverها و serviceها و آدرس فیزیکی گره را برای mediator فرستاده و خود را register می‌کند و او اطلاعات لازم برای دسترسی به سرویس‌های serverهای هر زیرسیستم را به شکل مناسب ذخیره می‌کند. همچنین `server`ها از طریق mediator در SateManager ثبت‌نام می‌کنند. همچنین هرگونه تغییر اطلاعات زیرسیستم‌ها به mediator و StateManager اطلاع داده می‌شود و آنها نیز به سایر زیرسیستم‌ها برای به روز رسانی blocklistها اطلاع می‌دهند (observer).

سناریو دوم: درخواست از یک زیرسیستم

یک client که در یکی از زیرسیستم‌های اصلی یا customer قرار دارد، سرویس مورد نظر را از پروکسی خود درخواست می‌کند؛ پروکسی درخواست را با اضافه نمودن اطلاعات کنترلی و pack کردن داده‌ها به broker محلی می‌فرستد؛ در صورتی که سرویس مربوط serverهای همان زیرسیستم باشد به server-side-proxy مربوطه ارجاع داده می‌شود و در غیراین‌صورت به bridge فرستاده می‌شود و او با پیاده سازی جزئیات مربوط به شبکه درخواست را به mediator می‌فرستد. عنصر mediator با بررسی درخواست، زیرسیستم های مورد نظر را پیدا کرده و به bridge آن درخواست را می‌فرستد (برای دریافت و فرستادن درخواست‌ها به زیرسیستم هدف و در mediator از اجزاء broker و bridge موجود در آن استفاده می‌شود). درخواست لایه به لایه تا server-side-proxy جلو می‌رود. در آنجا ابتدا درخواست unpack می‌شود و سپس checkAccess صدا زده می‌شود که در آن serverState.checkAccess صدا زده می‌شود و دسترسی به سرویس مورد نظر بررسی می‌شود. اگر وضعیت نرمال باشد که همیشه دسترسی برقرار است و سرویس از server صدا زده می‌شود و خروجی یا خطا لایه به لایه به عقب برمی‌گردد تا به کلاینت برسد. در وضعیت اضطراری نیز نام زیرسیستم مربوط به کلاینت در blocklistای که پروکسی در خودش نگه‌داری می‌کند بررسی شده و سرویس اجرا یا رد می‌شود.

سناریو سوم: تغییر وضعیت زیرسیستم ها و کنترل دسترسی ها در زمان اجرا

یک پردازه یا thread در زیرسیستم StateManager در زمان اجرا به طور مرتب وضعیت کلی سیستم و زیرسیستم‌ها را بررسی می‌کند و در صورت نیاز پیام تغییر وضعیت را از طریق فراخوانی setCriticalState یا setNormalState در خودش به زیرمجموعه ای از serverها می‌فرستد. این کار از طریق الگوی observer و صدا زدن متد notifyNewState در mediator انجام می‌شود. عنصر mediator زیرسیستم های مربوط به سرورهای ذکرشده را پیدا کرده و درخواست setState را به آنها می‌فرستد. وقتی درخواست بعد از چند لایه به server-side-proxy برسد، بعد از unpack کردن، متد setState خود را اجرا می‌کند که در آن serverState.setNextState اجرا می‌شود (یا خودش مستقیم state را ست می‌کند).

همچنین خود server-side-proxy با توجه به سابقه‌ی درخواست های رد شده از او و نتایج آنها، مثلا درخواست‌های ناموفق و خطا، می‌تواند state را تغییر دهد.

مشکلات راه حل پیشنهادی

- **کاهش کارایی:** به دلیل سطوح زیاد indirection که در الگوهای استفاده شده به خصوص broker وجود دارد، کارایی سیستم اندکی کاهش می‌یابد. البته همانطور که در قسمت زیرمسئله ها ذکر شد، در بعضی موارد از افزودن سطوح اضافه‌تر که ماهیت مسئله آن ها را می‌طلبید خودداری کردیم و یک trade-off بین کارایی و استفاده از الگو ها را تا حدی رعایت کردیم.
- **افزایش تعداد اشیا**: با افزودن تعداد زیادی کلاس به مسئله تعداد اشیا زیاد شده و این مورد در کارایی نیز تاثیر منفی دارد.
- **دشواری پیکربندی:** از این جهت که زیرسیستم ها نیاز به دانستن اطلاعات پیچیده درباره ی ارتباط با همه‌ی زیرسیستم‌ها نیستند و فقط با mediator پیکربندی می شوند ، دشواری پیکربندی کاهش یافته‌است اما پیکربندی serverها و clientها با broker و پیکربندی blocklistها، به سیستم پیچیدگی اضافه می‌کند و پیکربندی را سخت‌تر می‌کند.
- **کلاس چندکاره سنگین:** به دلیل استفاده از mediator پتانسیل ایجاد God Class وجود دارد.
- **Fault tolerance** پایین: از معایب استفاده از broker است که در صورت ایجاد مشکل در یکی از اجزای مسیر عبور درخواست‌ها و جواب‌ها، دچار خطا می‌شویم. همچنین single point of failure به دلیل mediator وجود دارد.
- **کاهش قابلیت test** و **debug**: در سطح زیرسیستم ها و واحد های زیردانه این مورد مزیت است اما در تست و دیباگ کل سیستم و مسیریابی خطا ها به دلیل زیاد بودن لایه و توزیع شده بودن بر بستر شبکه دچار مشکل هستیم.
- **وابستگی**: به دلیل نگهداری blocklistها در سمت proxyها ، یک وابستگی ایجاد کرده ایم که تغییر در نام زیرسیستم‌ها باید در این لیست نیز تغییر ایجاد کند. وابستگی mediator به زیرسیستم ها نیز زیاد است.
- **همزمانی**: طرح ارائه شده با فرض synchronous بودن درخواست‌ها و جواب‌ها ارائه شده‌است.با تغییراتی جزئی می‌توان asynchronous را نیز طراحی کرد.

سوال دوم: Java Idioms

مورد اول: توجه به کارایی در چسباندن رشته‌ها (آیتم ۶۳ از کتاب Effective Java)

همانطور که می‌دانیم، String در زبان جاوا به صورت immutable است. بنابراین وقتی از عملگر + برای چسباندن آنها به هم استفاده می‌کنیم، یک کپی از محتوای هر رشته ساخته می‌شود و در رشته‌ی نهایی قرار داده می‌شود. بنابراین در صورتی که n رشته را با این روش به هم بچسبانیم، الگوریتم مورد نظر به لحاظ زمانی از مرتبه‌ی $O(n^2)$ خواهد بود . مثال زیر از کتاب آورده شده و این شکل غلط را نشان می‌دهد. در این کد، یک متغیر شامل یک متن چند خطی داریم. تابع numItems نشان‌دهنده‌ی تعداد کل خط‌ها و lineForItem عدد i را به صورت پارامتر ورودی دریافت کرده و متن مربوط به خط iام را برمی‌گرداند. در هر تکرار از حلقه یک خط را خوانده و با + به خط های قبلی می‌چسبانیم و در متغیر result ذخیره می‌کنیم.

```
public String statement() {
   String result = "";
   for (int i = 0; i < numItems(); i++)
       result += lineForItem(i); // String concatenation
   return result;
}
```

راه حل‌های جایگزین: با استفاده از ۴ روش زیر می‌توان این مشکل کارایی را برطرف کرد:

۱. تلاش برای ارائه‌ی الگوریتمی که نیاز به چسباندن رشته‌ها ندارد و می‌توان تک به تک روی رشته‌ها و بدون چسباندن، عمل مورد نظر را انجام داد.
۲. استفاده از آرایه‌ی کاراکترها به جای String
۳. استفاده از StringBuilder به جای String: ابتدا یک نمونه از StringBuilder به اندازه کل کاراکترها ساخته شده و سپس به کمک تابع append آن، خط‌ها به هم چسبانده شده‌اند و در نهایت خروجی به شکل String برگردانده‌شده‌است. از ورژن ۶ جاوا تلاش‌های زیادی برای سریع شدن عملگر + روی رشته‌ها انجام‌شده‌است، اما همچنان تفاوت روش زیر و روش +، بسیار زیاد است. مثلا اگر در کد زیر، تعداد خطوط ۱۰۰ و تعداد کاراکترهای هر خط ۸۰ باشد، روش StringBuilder روی یک ماشین (ماشین نگارنده ی کتاب) حدود ۶,۵ برابر سریع تر از روش + است. زیرا یکی خطی و دیگری درجه دوم است و هرچه تعداد کاراکترها بیشتر شود این اختلاف

بیشتر می‌شود. در صورتی که از یک StringBuilder بدون ست کردن سایز رشته‌ی نهایی در Constructor و با سایز پیش‌فرض آن استفاده شود، عدد ۶,۵ به ۵,۵ کاهش می‌یابد.

```
public String statement() {
   StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
   for (int i = 0; i < numItems(); i++)
       b.append(lineForItem(i));
   return b.toString();
}
```

۴. استفاده از StringBuffer به جای String

StringBuffer نیز همانند StringBuilder جایگزین String است و نحوه‌ی استفاده از آنها شبیه به هم است، اما مزیت آن نسبت به StringBuilder، پشتیبانی از Synchronization و thread-safe بودن آن است.

بنابراین در استفاده از + در چسباندن رشته‌ها در زبان جاوا باید نگران کارایی بود و فقط در مواردی مورد استفاده قرار گیرد که تعداد بسیار کمی رشته را به هم بچسبانیم. در غیر این صورت باید از روش‌های جایگزین ذکر شده استفاده نمود.

مورد دوم: رها نکردن exceptionها (آیتم ۷۷ از کتاب Effective Java)

رها کردن exceptionها بدون مدیریت آنها، زیر سوال بردن مفهوم و فلسفه وجودی آنهاست. وقتی که برنامه نویس در طراحی یک قسمت از برنامه یا API ، برای برخی شرایط، exception درنظرگرفته و به اصطلاح آن را throw کرده است، حتما جوانبی را سنجیده که ممکن است دچار خطایی شویم که در ادامه‌ی برنامه مشکل ساز است. بنابراین گذرکردن از کنار آنها بدون انجام actionهای لازم مانند نادیده گرفتن یک زنگ خطر برای وجود یک حادثه در آینده است. منظور از رها کردن و نادیده گرفتن خالی گذاشتن بلاک catch است، مانند مثال زیر:

```
try {
   //code
} catch (SomeException e) {
}
```

مثال بارز این مسئله در دنیای واقعی این است که در هنگام قطع کردن زنگ خطر آتش سوزی، زنگ را قطع کنیم و اجازه ندهیم دیگران نیز صدا را بشنوند. ممکن است هیچ اتفاقی در آینده نیفتد اما این امکان نیز وجود دارد که آتش سوزی رخ دهد و نتوانیم به موقع اقدامات لازم را انجام دهیم. بنابراین تا جایی که می‌توان نباید بلاک catch را خالی گذاشت.

اما مسئله اینجاست که گاهی اوقات مجبور به خالی گذاشتن بلاک می‌شویم. برنامه‌نویسی که exception را در نوشته، نمی داند که API او در کجا استفاده می‌شود و باید برای تمام حالات ممکن برنامه ریزی کرده باشد. بنابراین ممکن است در موردی که ما استفاده می‌کنیم، اکسپشن پرتاب شده قابل نادیده‌گرفتن باشد. به طور مثال کانکشن‌های پایگاه داده و FileInputStream. مثلا در بستن یک FileInputStream در صورت بروز مشکل exception پرتاب می‌شود اما اگر ما وضعیت فایل را تغییر نمی‌دهیم و فقط از فایل اطلاعات می‌خوانیم، نیازی به نوشتن action برای بازیابی فایل و مدیریت آن نیست و برنامه ما می‌تواند ادامه داشته باشد. اما بهتر است که یک log در بلاک catch بگذاریم تا در صورت تکرار زیاد این اتفاق دلیل آن را بررسی کنیم.

در صورتی که تصمیم به خالی گذاشتن بلاک catch گرفتیم، باید حتما یک کامنت در آنجا بگذاریم و دلیل این کار را بیان کنیم و نام متغیر اکسپشن را ignored بگذاریم. همانطور که گفته شد، log کردن نیز مفید است.

در قطعه کد زیر نمونه‌ای از این کار را مشاهده می‌کنیم:

```
Future<Integer> f = exec.submit(planarMap::chromaticNumber);
int numColors = 4; // Default; guaranteed sufficient for any map
try {
   numColors = f.get(1L, TimeUnit.SECONDS);
} catch (TimeoutException | ExecutionException ignored) {
   // Use default: minimal coloring is desirable, not required
}
```

همانطور که در کد دیده می‌شود، یک مقدار پیش‌فرض برای متغیر numColors در نظر گرفته شده است که ایده‌آل است و مشکلی در ادامه کد ایجاد نمی‌کند، بنابراین اگر کد درون try، یکی از exception های ذکر شده را پرتاب کند و مقدار جدید متغیر numColors قابل ست شدن نباشد، مشکلی پیش نمی‌آید و ادامه‌ی برنامه قابل اجراست. بنابراین از روش گفته شده استفاده شده است و بلاک catch خالی است و از کامنت استفاده شده و متغیر با نام ignored تعریف شده است.

این روش برای انواع checked و unchecked از exception قابل اجراست. اما به طور کلی باز هم می‌گوییم که از رها کردن exceptionها استفاده نکنید، حتی اگر یک exception قابل پیش‌بینی یا یک خطای

برنامه‌نویسی باشد. مدیریت آنها، خطایابی و دیباگ را ساده تر می‌کند، زیرا در صورت رها کردن آنها، ممکن است در ادامه‌ی اجرای برنامه و در جایی که هیچ ارتباطی به آن مشکل ندارد و اصلا انتظارش را نداریم، کد دچار خطا شود و بدین صورت خطایابی آن دشوار می‌شود و نمی توان منبع خطا را به راحتی پیدا کرد.

منابع

۱. اسلایدها و مطالب تدریس شده در کلاس

۲. نمونه تمرین‌های ترم‌های گذشته در سایت درس

3. http://www.openloop.com/softwareEngineering/patterns/architecturePattern/arch_Broker.htm
4. J. Bloch. Effective Java. Addison-Wesley, 2017