دانشگاه تهران

پردیس دانشکده‌های فنی

دانشکدهٔ مهندسی برق و کامپیوتر

طراحی منطق دامنه بر اساس تبادل ناهمگام پیغام

نگارش

وحید ذوقی شال

استاد راهنما

دکتر رامتین خسروی

پایان‌نامه برای دریافت درجهٔ کارشناسی ارشد در رشتهٔ

مهندسی کامپیوتر - گرایش نرم‌افزار

شهریور ۱۳۹۱

تقدیم به پدرم، مادرم و همسر مهربانم

قدردانی

در ابتدا لازم می‌دانم از جناب آقای دکتر رامتین خسروی که در انجام این پژوهش افتخار استفاده از راهنمایی ایشان را داشتم، تشکر و قدردانی کنم. مطمئناً این کار بدون کمک‌های همه‌جانبه و بی‌شائبه‌ی ایشان امکان‌پذیر نبود. از اعضای هیئت داوران محترم نیز برای فرصتی که در اختیار من قرار دادند تشکر می‌کنم.

طراحی منطق دامنه بر اساس تبادل ناهمگام پیغام

چکیده

در سال‌های اخیر گرایش به مدل اکتور چه در دنیای پژوهش و چه در صنعت افزایش پیدا کرده است. تغییر روند افزایش سرعت پردازنده‌ها به سمت افزایش تعداد هسته‌ها، استفاده از زیرساخت‌های محاسبات ابری و گرایش به تولید برنامه‌های توزیع شده می‌توانند از جمله‌ی دلایل این علاقه‌مندی باشند. از سوی دیگر علیرغم وجود منابع گسترده برای یادگیری طراحی به روش شیءگرا، کمبود پژوهش در زمینه‌ی روش‌ها و نکات موجود در طراحی شیءگرای همروند محسوس می‌باشد. در این پژوهش تلاش شده است تا با انجام طراحی یک سیستم انتخاب شده با استفاده از تبادل ناهمگام پیغام، روش‌ها، الگوها و نکات موجود در این روش طراحی بررسی شده و به صورت قابل استفاده‌ای ارائه گردند. طراحی انجام شده با استفاده از معیارهای کیفی نرم‌افزار، با طراحی شیءگرای عادی (ترتیبی) مقایسه شده و نشان داده شده است که از نظر کیفی این طراحی قابل مقایسه و در مواردی بهتر از طراحی ترتیبی است. علاوه بر این با استفاده از این نوع طراحی، همروندی ذاتی در سیستم ایجاد می‌شود و قابلیت توزیع برنامه به دلیل خصوصیات معنایی مدل اکتور به صورت قابل توجهی افزایش می‌یابد.

واژه‌های کلیدی: *طراحی منطق دامنه، تبادل ناهمگام پیغام، مدل اکتور، همروندی*

فهرست مطالب

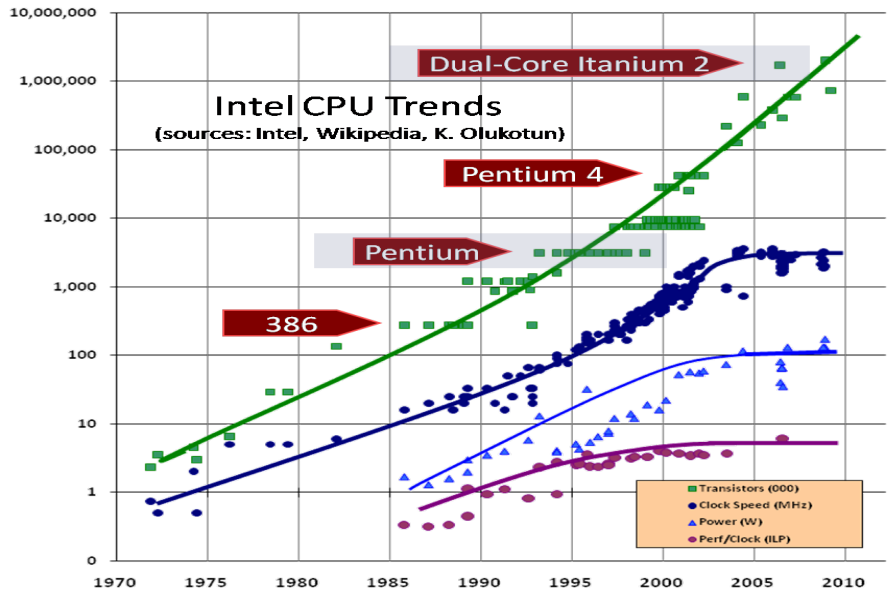
فهرست تصاویر

فصل ۱

مقدمه

۱.۱ انگیزه‌ی پژوهش و صورت مسئله

در سال‌های اخیر در روند افزایش سرعت پردازنده‌ها تغییر قابل ملاحظه‌ای به وجود آمده است. در سالهای گذشته افزایش سرعت پردازنده‌ها به معنی افزایش فرکانس چیپ‌های پردازنده بوده است، بدین مفهوم که تقریبا با گذشت هر ۲ سال، سرعت پردازشی پردازنده‌ها حدودا ۱/۵ برابر شده‌اند. این روند در شکل ۱.۱ تا حدود سال ۲۰۰۵ قابل مشاهده است. در این روند شاهد افزایش بدون توقف سرعت پردازنده‌ها بوده‌ایم. همان طور که شکل نشان می‌دهد، در ادامه‌ی این این روند شاهد توقف افزایش سرعت پردازش بوده‌ایم. با وجود این توقف افزایش سرعت، تعداد ترانزیستورهای پردازنده‌ها طبق روند قبلی افزایش یافته است. این تغییر به این معناست که در زمینه‌ی قدرت پردازش پردازنده‌ها، افزایش تعداد هسته‌های پردازشی جایگزین افزایش سرعت پردازشی هسته‌ها شده است. با توجه به این تغییر در روند بهبود سرعت پردازنده‌ها، نقش طراحی برنامه در افزایش کارایی آن از نظر سرعت اجرا پررنگ‌تر شده است. در این وضعیت عامل اصلی تاثیر گذار بر سرعت اجرای برنامه، تعداد فرایندهای همروند آن می‌باشد. با افزایش همروندی، کارایی سیستم می‌تواند تا میزان محدودی افزایش پیدا کند. رویکرد معمول برای افزایش همروندی سیستمها که در پژوهش‌های متعددی به آن پرداخته شده است اختصاص فرایندها یا ریسمان های همروند برای انجام محاسبات مشابه می‌باشد. به عنوان مثال در یك برنامه تحت وب، تمام پردازش‌های مربوط به یك درخواست که به یك وب سرور فرستاده می‌شود در یك ریسمان سرور اجرا می‌شود. در این رویکرد برای افزایش کارایی بیشتر تمرکز روی تنظیم تعداد ریسمان‌های سرویس دهنده و نیز بهینه



شکل ۱.۱: روند افزایش سرعت پردازنده‌ها در سال‌های اخیر

کردن زمانبندی تراکنشهای پایگاه داده می‌باشد و بخشی از منطق دامنه که برای سرویس دادن به درخواست اجرا می‌شود تاثیر چندانی بر کارایی ندارد. حوزه بحث این پژوهش، طراحی منطق دامنه برنامه مبتنی بر تبادل ناهمگام پیغام می‌باشد. ارتباط ناهمگام بین اشیاء برنامه منجر به ایجاد همروندی ریزدانه می‌گردد. در همروندی ریزدانه که در این پژوهش به آن پرداخته خواهد شد، همروندی به عنوان خاصیتی در طراحی منطق دامنه در نظر گرفته می‌شود. در این رویکرد برای افزایش همروندی، به جای افزایش تعداد ریسمان‌هایی که هر کدام یك کار مشابه را از ابتدا تا انتها انجام می‌دهند، منطق پردازش یك درخواست با استفاده از ارتباط ناهمگام اشیاء و همروندی ریزدانه طراحی می‌شود. برای پیاده‌سازی همروندی به جای استفاده از ریسمان‌ها، از روش تبادل ناهمگام پیغام استفاده خواهد شد. دلیل عدم استفاده از ریسمان‌ها یکی مشکالت ناشی از وجود حالت مشترك [۱] بین ریسمان‌ها است و دیگری این است که آمیخته شدن کدهای مربوط به ریسمان با منطق دامنه‌ی برنامه مطلوب نمی‌باشد. هدف از انجام این پژوهش بررسی روش طراحی شیءگرای منطق دامنه بر اساس ایده‌ی ارتباط ناهمگام و همروندی ریزدانه و بررسی اثر آن در ویژگیهای کیفی نرم‌افزار است. در این پژوهش، ویژگی‌های کیفی کارایی [۲] و تغییرپذیری [۳] برای بررسی انتخاب شده‌اند.

[۱] Shared State

[۲] Performance

[۳] Modifiability

۲.۱ خلاصه‌ی دستاوردهای پژوهش

برخی از دستاوردهای این پژوهش را می‌توان به این ترتیب برشمرد:

- یک سیستم نمونه انتخاب شده و طراحی منطق دامنه‌ی آن به روش تبادل ناهمگام پیغام به طور کامل انجام شده است. ارائه‌ی روش طراحی به صورت مرحله‌ای و افزایشی باعث شده است تا بتوان از آن به صورت دستورالعملی برای طراحی همروند استفاده کرد.
- خروجی مهم پژوهش، روش‌ها و الگوهایی است که در این نوع طراحی کاربرد دارد. در هر الگوی استخراج شده، روش پیاده‌سازی در مدل اکتور و کاربردهای الگو از نظر منطق دامنه بررسی شده است.
- تجربیاتی که در طراحی‌های صورت گرفته کسب شده به صورت قابل استفاده‌ای ارائه شده است و مطالعه‌ی این تجربیات، خواننده را با نکات ظریف و حساسی آشنا می‌کند که انجام طراحی به روش تبادل ناهمگام پیغام را بسیار ساده‌تر می‌کند.
- در ارزیابی روش طراحی ناهمگام، خصوصیات کیفی این روش از جمله تغییرپذیری و کارایی آن با روش طراحی شیءگرای ترتیبی مقایسه شده و نشان داده شده است که علاوه بر اینکه از نظر تغییرپذیری دو روش قابل مقایسه هستند، طراحی به روش تبادل ناهمگام پیغام در مواردی باعث افزایش چشم‌گیر کارایی سیستم می‌گردد.

۳.۱ ساختار پایان‌نامه

برای بررسی این موارد، ساختار این متن در ۵ فصل تنظیم گردیده است:

- فصل ۲ به ارائه‌ی برخی پیش‌نیازهای طراحی به روش تبادل ناهمگام پیغام می‌پردازد. مدل اکتور و کتابخانه‌ی اکتور اسکالا در این فصل معرفی شده‌اند.
- در فصل ۳ پژوهش‌های مرتبط معرفی شده‌اند. الگوهای طراحی با اکتورها و نیز کاربردهای صنعتی رویکرد تبادل ناهمگام پیغام در این فصل بررسی شده‌اند. علاوه بر آن روش‌های طراحی منطق دامنه در برنامه‌نویسی شیءگرا به طور مختصر معرفی شده‌اند.
- در فصل ۴ روش طراحی منطق دامنه با استفاده از تبادل ناهمگام مورد بررسی قرار گرفته. روش طراحی با انتخاب

یک سیستم نمونه و بسط روش طراحی آن ارائه شده و در پایان الگوها و نکات مهم طراحی استخراج شده‌اند. علاوه بر این، معیارهای کیفی سیستم طراحی شده با رویکرد تبادل ناهمگام بررسی شده و با همین ویژگی‌ها در رویکرد طراحی ترتیبی مقایسه شده است.

- نهایتا فصل ۵ به جمع‌بندی پژوهش، ارائه‌ی دستاوردها و ذکر تعدادی از جهت‌گیری‌های مرتبط برای پژوهش‌های آینده می‌پردازد.

فصل ۲

پیش‌زمینه تحقیق

در این فصل به طور اجمالی مروری بر پیش‌زمینه‌ی پژوهش انجام شده است. در هر بخش سعی شده است که با حفظ اختصار، تنها جنبه‌های کاربردی مرتبط با پژوهش مطرح گردد.

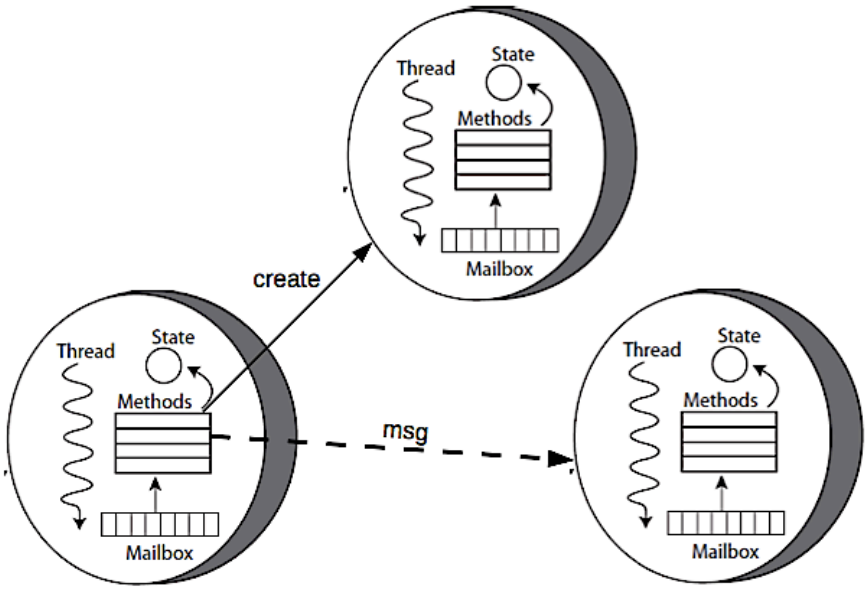
۱.۲ مدل اکتور

در زمینه‌ی برنامه‌نویسی همروند پژوهش‌های مختلفی صورت گرفته و مدل‌هایی ارائه شده است[۱]. در این میان مدل اکتور[۱] با توجه به استفاده از ارتباط ناهمگام و قابلیت توزیع بالا توجه زیادی را به خود جذب کرده است. با توجه به ارتباط تنگاتنگ این مدل با پژوهش حاضر، در این بخش به معرفی اجمالی این مدل می‌پردازیم. کلمه‌ی اکتور برای اولین بار در حدود ۳ دهه پیش توسط هیوئیت [۲] به کار گرفته شد. اکتور در کاربرد هیوئیت به معنی موجودیت‌های فعالی بود که در یک پایگاه دانش به جستجو پرداخته و در نتیجه کنش‌هایی را ایجاد می‌نمودند. در دهه‌های بعدی گروه هیوئیت با تکیه بر اکتورها به عنوان عامل‌های محاسباتی[۲] مدل اکتور را به عنوان یک مدل محاسباتی همروند گسترش داد. خلاصه‌ای از تاریخچه‌ی مدل اکتور در [۳] موجود است. امروزه برداشت عمومی از مدل اکتور مربوط به آقا[۴] می‌باشد. در ادامه‌ی این بخش مشخصات مدل اکتور ارائه شده است.

---

[۱] Actor Model

[۲] agents of computation



شکل ۱.۲: اکتورها موجودیت‌های همروندی هستند که به صورت ناهمگام تبادل پیغام انجام می‌دهند.

مدل اکتور که توسط هیوئیت و آقا [۲، ۵، ۶] ایجاد شده‌است، یک نمایش سطح بالا از سیستم‌های توزیع‌شده فراهم می‌کند. اکتورهااشیای لفافه‌بندی‌شده‌ای هستند که به صورت همروند فعالیت می‌کنند و دارای رفتار[۳] قابل تغییر هستند. اکتورها حالت مشترک[۴] ندارند و تنها راه ارتباط بین آنها تبادل ناهمگام پیغام است. در مدل اکتور فرضی در مورد مسیر پیغام و میزان تاخیر در رسیدن پیغام وجود ندارد، در نتیجه ترتیب رسیدن پیغام‌ها غیرقطعی است. در یک دیدگاه می‌توان اکتور را یک شیء در نظر گرفت که به یک ریسمان‌ریسمان[۵] کنترل، یک صندوق پست و یک نام غیر قابل تغییر و به صورت سرارسی یکتا [۶] مجهز شده است. برای ارسال پیغام به یک اکتور، از نام آن استفاده می‌شود. در این مدل، نام یک اکتور را می‌توان در قالب پیغام ارسال کرد. پاسخگویی به هر پیام شامل برداشتن آن پیام از صندوق پستی و اجرای عملیات متناسب با آن است. این اجرای عملیات به صورت تجزیه‌ناپذیر[۷] و بی‌وقفه خواهد بود[۴].

همان گونه که گفته‌شد، مدل اکتور سیستم را در سطح بالایی از انتزاع مدل می‌کند. این ویژگی دامنهٔ سیستم‌های

[۳] Behavior

[۴] Shared State

[۵] Thread

[۶] Globally Unique

[۷] Atomic

قابل مدلسازی توسط مدل اکتور را بسیار وسیع نموده‌است. انواع سیستم‌های سخت‌افزاری و نرم‌افزاری طراحی‌شده برای زیرساخت‌های خاص یا عام، و همچنین الگوریتم‌ها و پروتکل‌های توزیع‌شدهٔ مورد استفاده در شبکه‌های ارتباطی از جملهٔ موارد مناسب برای بهره‌گیری از مدل اکتور هستند. علاوه بر این، خصوصیت تبادل ناهمگام پیغام، باعث می‌شود مدل اکتور برای مدل کردن سیستم‌های توزیع شده و متحرک بسیار ایده‌آل باشد[۷]. شکل ۱.۲ شمای کلی از مدل اکتور و نحوه‌ی تعامل اکتورها را نشان می‌دهد.

یک اکتور در نتیجه‌ی دریافت پیغام احتمالا محاسباتی انجام می‌دهد و در نتیجه‌ی آن یک از ۳ عمل زیر را انجام می‌دهد:

- ارسال پیغام به سایر اکتورها
- ایجاد اکتور جدید
- تغییر حالت محلی

۱.۱.۲ معناشناسی

[۸] از نظر معناشناسی مشخصه‌های کلیدی مدل محض اکتور عبارتند از: لفافه‌بندی و تجزیه‌ناپذیری[۹]، انصاف[۱۰]، استقلال از مکان[۱۱]، توزیع[۱۲] و تحرک[۱۳] [۷]. باید توجه داشت که این مشخصه‌ها در مدل محض وجود دارند و این الزاما به این معنی نیست که تمام زبان‌های مبتنی بر مدل اکتور از این مشخصه‌ها پشتیبانی می‌کنند. ممکن است تعدادی از این مشخصه‌ها در زبان‌های مبتنی بر اکتور با در نظر گرفتن اهدفی مانند کارایی و سهولت پیاده‌سازی نشده‌اند. در این موارد باید با به کار بردن ابزارهای بررسی ایستا، مترجم‌ها و یا با تکیه بر عملکرد درست برنامه‌نویس از صحت عملکرد برنامه اطمینان حاصل کرد [۸].

---

[۸] Semantics

[۹] Encapsulation and Atomicity

[۱۰] Fairness

[۱۱] Location Transparency

[۱۲] Distribution

[۱۳] Mobility

- **لفافه‌بندی و تجزیه‌ناپذیری:**[۱۴] نتیجه‌ی مستقیم مشخصه‌ی لفافه‌بندی در اکتورها این است که درهیچ دو اکتوری، به اشتراک گذاری حالت وجود ندارد. این مشخصه، تجزیه‌ی شیءگونه‌ی برنامه را تسهیل می‌کند. در زبان‌های برنامه‌نویسی شیء-بنیان مشخصه منجر به ایجاد تغییر تجزیه‌ناپذیر شده است. به این صورت که وقتی یک شیء، شیء دیگری را فراخوانی می‌کند، شیء مقصد تا پایان محاسبات مربوط به این فراخوانی، به فراخوانی‌های دیگر پاسخ نمی‌دهد. این مشخصه به ما اجازه می‌دهد تا بتوانیم در باره‌ی رفتار یک شیء در قبال دریافت یک پیغام (فراخوانی) با توجه به حالت شیء در زمان دریافت آن استدلال کنیم.

در محاسبات همروند، وقتی یک اکتور مشغول انجام محاسبات مربوط به یک پیغام است، امکان دریافت پیغام جدید توسط آن وجود دارد اما مشخصه‌ی تجزیه‌ناپذیری تضمین می‌کند که پیغام جدید امکان قطع محاسبات جاری اکتور و تغییر حالت محلی آن را ندارد. این مشخصه الزام می‌کند که اکتور گیرنده، در هر لحظه فقط یک پیغام در حال پردازش داشته باشد و محاسبات مربوط به پیغام جاری را در یک قدم بزرگ[۱۵] به صورت تجزیه ناپذیر طی کند. [۳] مشخصه‌های معناشناسی لفافه‌بندی و تجزیه ناپذیری به طور چشم‌گیری از عدم قطعیت مدل اکتور می‌کاهند و با کوچکتر کردن فضای حالت برنامه‌های نوشته شده در مدل اکتور، این برنامه‌ها را برای استفاده در ابزارهای آزمون درستی و verification(؟) قابل استفاده می‌کند[۹]. این دو مشخصه مجموعا باعث می‌شوند تا بتوانیم بر اساس پیغام انتخاب شده برای اجرا و وضعیت محلی اکتور در هنگام شروع به اجرا ، رفتار یک اکتور قابل پیش‌بینی باشد.

- **انصاف:** انصاف در مدل اکتور به این مفهوم است که پیغام فرستاده شده نهایتا به اکتور مقصد خواهد رسید، مگر آنکه اکتور مقصد به طور دائمی غیر فعال شده باشد. لازم به ذکر است که این تعریف از انصاف در رسیدن پیغام به اکتور مقصد، متضمن انصاف در زمان‌بندی اکتورها است. به این مفهوم که در صورتی که یک اکتور در اثر زمان‌بندی غیر منصفانه، موفق به اخذ نوبت اجرا نشود، پیغام‌های فرستاده شده به مقصد آن اکتور هرگز به مقصد نخواهند رسید. انصاف علاوه بر تضمین رسیدن پیغام‌ها، امکان استدلال مناسب درباره‌ی نحوه‌ی تداوم اجرای برنامه[۱۶] را فراهم می‌کند. میزان طبیعتا میزان موفقیت در تضمین این مشخصه در محیط‌های مبتنی بر اکتور وابسته به منابع موجود در سیستم در حال اجرا است [۸].

- **استقلال از مکان، توزیع و تحرک:** در مدل اکتور، ارسال پیغام به یک اکتور تنها از طریق دسترسی به نام آن

[۱۴] Encapsulation and Atomicity

[۱۵] Macro-Step

[۱۶] Liveness Property

اکتور ممکن می‌شود. مکان واقعی اکتور تأثیری روی نام آن ندارد. هر اکتور دارای فضای آدرس مربوط به خود است که می‌تواند کاملا متفاوت با دیگر اکتورها باشد. اکتورهایی که به یکدیگر پیغام می‌فرستند می‌توانند روی یک هسته از یک پردازنده‌ی مشترک اجرا شوند یا اینکه در ماشین دیگری که از طریق شبکه به آنها مرتبط می‌شوند در حال اجرا باشند. مشخصه‌ی استقلال از مکان در مدل اکتور به برنامه‌نویس این امکان را می‌دهد که فارغ از نگرانی درباره‌ی محل اجرای اکتور ها به برنامه‌نویسی بپردازد. عدم اطلاع از مکان اجرای اکتوران منجر به قابلیت حرکت در آنها می‌شود. تحرک به صورت قابلیت انتقال پردازش به نودهای دیگر تعریف می‌شود.در سطح سیستم، تحرک از جهت توزین بار [۱۷]، قابلیت تحمل خطا[۱۸] و نیز پیکربندی مجدد[۱۹] حائز اهمیت است. پژوهش‌های پیشین نشان می‌دهد که قابلیت تحرک در رسیدن به کارایی مقیاس‌پذیر به ویژه در کاربردهای بی‌قاعده[۲۰] روی ساختار داده‌های پراکنده مفید است[۱۰]. در کاربردهای دیگر، توزیع بهینه به شرایط زمان اجرا و میزان بار وابسته است. به عنوان مثال، در کاربردهای وب، تحرک با توجه به شرایط شبکه و امکانات کلاینت مورد استفاده قرار می‌گیرد[۱۱]. از سوی دیگر، قابلیت تحرک می‌تواند در کاهش انرژی مصرفی در اثر اجرای کاربردهای موازی مفید باشد. در این کاربردها، محاسبات موازی به صورت پویا بین تعداد هسته‌های بهینه (تعداد هسته‌هایی که منجر به کمترین مصرف می‌شوند) توزین می‌شوند. قسمت‌های مختلف یک کاربرد می‌تواند شامل الگوریتم‌های موازی مختلفی باشد و میزان مصرف انرژی یک الگوریتم به تعداد هسته‌های مشغول اجرای الگوریتم و نیز بسامد اجرای آن هسته‌ها بستگی دارد[۱۲]. در نتیجه، ویژگی تحرک پذیری اکتورها، ویژگی مهمی برای برنامه نویسی در معماری‌های چند-هسته‌ای به شمار می‌آید.

۲.۱.۲ پیاده‌سازی‌ها

برای مدل اکتور زبان‌ها و چارچوب‌های زیادی توسعه داده شده است. ABCL، POOL، ConcurrentSmalltalk و CEiffel ،ACT++ تعدادی از پیاده‌سازی‌های اولیه از این مدل می‌باشند. مرجع [۱] به بررسی این زبان‌ها پرداخته است. شاید بتوان زبان ارلانگ[۲۱][۱۳] را معروفترین پیاده‌سازی مدل اکتور دانست. این زبان در حدود ۲۲ سال قبل

[۱۷] Load-Balancing

[۱۸] Fault Tolerance

[۱۹] Reconfiguration

[۲۰] Irregular

[۲۱] Erlang

برای برنامه‌نویسی سوئیچ‌های مخابراتی شرکت اریکسون[۲۲] توسعه داده شد. علاوه بر ارلانگ زبان‌ها و چارچوب‌های مبتنی بر مدل اکتور دیگری نیز در سال‌های اخیر مورد استفاده گرفته‌اند که کتابخانه‌ی اکتور اسکالا [۲۳] [۱۴]، Ptolemy [۱۵]، SALSA [۱۶]، CHARM++ [۱۷]، ActorFoundry [۱۸]، Library Agents Asynchronous [۱۹] از جمله‌ی آنها هستند. از کاربردهای متن-باز که بر مبنای مدل اکتور توسعه داده شده‌اند می‌توان به سیستم تبادل پیغام توئیتر[۲۴] و چارچوب تحت وب لیفت[۲۵] و از میان کاربردهای تجاری می‌توان به سیستم گپ[۲۶] فیسبوک و موتور بازی وندتا[۲۷] اشاره کرد. در این پژوهش برای پیاده‌سازی نسخه‌ی مبتنی بر تبادل ناهمگام پیغام از کتابخانه‌ی اکتور اسکالا استفاده شده است.

۲.۲ معرفی زبان اسکالا و کتابخانه‌ی اکتور اسکالا

همان طور که در بخش ۲.۱.۲ اشاره شد، پیاده‌سازی‌های مختلفی از مدل اکتور در زبان‌ها و چارچوب‌های برنامه‌نویسی ارائه شده است. مقاله‌ی [۸] به بررسی و مقایسه‌ی این پیاده‌سازی‌ها پرداخته است. در این پژوهش زبان اسکالا و کتابخانه‌ی اکتور آن برای پیاده‌سازی مطالعه‌ی موردی انتخاب شده است. گستردگی ابزار و همچنین فعال بودن جامعه[۲۸]ی برنامه‌نویسی این زبان اصلی‌ترین انگیزه‌های انتخاب این زبان برای پیاده‌سازی بوده‌اند. ضمنا با توجه به انتخاب زبان جاوا برای پیاده‌سازی نسخه‌ی متداول مورد مطالعه و ارتباط تنگاتنگ زبانهای اسکالا و جاوا، انتخاب زبان اسکالا منجر به سهولت ارزیابی مقایسه‌ای مطالعه‌ی موردی شده است. در این بخش به معرفی اجمالی زبان اسکالا و کتابخانه‌ی اکتور آن پرداخته شده است. هدف از این معرفی، سهولت درک روش طراحی پیشنهادی در فصل ۳ می باشد و به همین دلیل از توضیح جزئیات و امکانات اضافی این زبان خودداری شده است. کتاب [۲۰] به عنوان منبع اصلی این بخش استفاده شده است.

---

[۲۲] Ericsson

[۲۳] Scala Actor Library

[۲۴] Twitter

[۲۵] Lift

[۲۶] Chat

[۲۷] Vendetta game engine

[۲۸] Community

```
1 class Course(var id: String, var name: String, var units: Int,
2   var preRequisites: List[Course]) extends BaseDomain {
3
4   override def equals(other: Any): Boolean =
5     other match {
6       case that: Course =>
7           id == that.id
8       case _ => false
9     }
10
11  def printPrerequisites() = {
12    for (pre <- preRequisites)
13      println(pre)
14  }
15
16  override def toString = "[id= " + id + ",name=" + name + ",units=" + units + "]"
17 }
```

شکل ۲.۲: قطعه کد نمونه برای زبان اسکالا

۱.۲.۲ زبان اسکالا

اسکالا مخفف عبارت *"زبان مقیاس‌پذیر"*[۲۹] است و اشاره به این نکته دارد که اسکالا برای رشد بر اساس نیاز کاربر طراحی شده است. اسکالا را می‌توان برای گستره‌ی وسیعی از کاربردها از نوشتن اسکریپت‌های کوچک گرفته تا پیاده‌سازی سیستم‌های بزرگ به کار برد. برنامه‌های اسکالا بر روی محیط اجرایی جاوا[۳۰] قابل اجرا هستند و در برنامه‌های اسکالا می‌توان از کتابخانه‌های استاندارد جاوا استفاده کرد. زبان اسکالا ترکیبی از ویژگی‌های زبان‌های تابعی و شیءگرا را در خود دارد. در زبان‌های تابعی، توابع مانند انواع داده‌ها قابل ارجاع هستند. اسکالا مانند جاوا دارای بررسی گونه‌های ایستا است.

در ادامه مشخصات نحوی زبان اسکالا در قالب یک مثال توضیح داده می‌شود. در شکل ۲.۲ قطعه کد اسکالا مربوط به کلاس Course نمایش داده شده است. برای آشنایی با نحو زبان اسکالا به بررسی این کد می‌پردازیم:

[۲۹] Scalable Language

[۳۰] JRE

در خطوط ۱ و ۲ کلاس Course و متغیرهای id، name، units و prerequisites به عنوان فیلدهای آن تعریف شده‌اند. در خط ۴ تابع equals از این کلاس override شده است. در اسکالا همانند جاوا هر کلاس به طور پیش‌فرض دارای یک تابع equals است که در صورت لزوم می‌توان آن را override کرد. همان‌طور که در کد مشخص است، تعریف تابع در اسکالا با کلمه‌ی کلیدی **def** انجام می‌گیرد. در خطوط ۴ تا ۸ شرط لازم برای یکسان بودن یک شیء از نوع Course با شیء حاضر پیاده‌سازی شده است. نوع و مقدار یک متغیر را می‌توان با استفاده از دستور .. case .. match با انواع و مقادیر دلخواه مقایسه کرد. نتیجه‌ی دستورات خطوط ۶ و ۷ این است که اگر متغیر other از نوع Course باشد و مقدار فیلد id آن با مقدار فیلد id از شیء حاضر یکسان باشد تابع مقدار true را برمی‌گرداند. خط ۸ به این معنا است که اگر هر حالت دیگری به جز حالت قبل بود مقدار false برگردانده می‌شود. در خط ۱۲ نمونه‌ای از حلقه‌ی for نمایش داده شده است. در اسکالا حلقه‌ها به صورت‌های متنوعی می‌توانند بیان شوند که در این مثال یک حالت از آنها نمایش داده شده است. در خط ۱۲ متغیر pre برای گرفتن مقدار موقت حلقه تعریف شده است. نکته‌ی جالب توجه این است که در این خط، نوع متغیر تعریف نشده است. در بخش قبل ذکر شد که اسکالا دارای خاصیت بررسی گونه‌های ایستا [۳۱] است. ظاهرا این دو امر در تناقض با یکدیگر هستند اما باید توجه داشت که در زبان اسکالا نوعی از استنتاجِ گونه[۳۲] در زمان ترجمه اتفاق می‌افتد. در این مورد با توجه به اینکه متغیر pre از لیست prerequisites مقداردهی می‌شود، گونه‌ی آن در زمان ترجمه قابل استنتاج است. خط ۱۶ تابع دیگری را نشان می‌دهد که در آن تابع override toString، شده است. نکته‌ی قابل توجه در مورد این قسمت از کد عدم استفاده از علامت { } برای تعیین حوزه‌ی تابع است. در زبان اسکالا به دلیل وجود ویژگی‌های زبان‌های تابعی، می‌توانیم با توابع مانند متغیرها و داده‌ها رفتار کنیم که این بخش از کد مثالی از این ویژگی است. همانطور که در این مثال مشخص است، در زبان اسکالا استفاده از نقطه‌ویرگول (؛) در اکثر موارد اختیاری است.

۲.۲.۲ کتابخانه‌ی اکتور اسکالا

همانطور که در بخش ۲.۱.۲ اشاره شد، یکی از پیاده‌سازی‌های مدل اکتور، کتابخانه‌ی اکتور اسکالا است. در این بخش به معرفی اجمالی کتابخانه‌ی اکتور اسکالا و طرز استفاده از آن برای برنامه‌نویسی همروند می‌پردازیم.

[۳۱] static type checking

[۳۲] type inference

```scala
import scala.actors._
object SillyActor extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println("I'm acting!")
      Thread.sleep(1000)
    }
  }
}
```

شکل ۳.۲: کد یک اکتور ساده در زبان اسکالا

۱.۲.۲.۲ ایجاد اکتور

اکتورها در اسکالا از کلاس scala.actors.Actor مشتق می‌شوند. شکل ۳.۲ کد مربوط به یک اکتور ساده را نشان می‌دهد. این اکتور کاری به صندوق پیغام‌ها ندارد و صرفا پنج بار پیغام !I'm acting را چاپ می‌کند و سپس اجرای آن خاتمه می‌یابد.

اکتورها در اسکالا با دستور ()start شروع به فعالیت می‌کنند. با شروع به فعالیت یک اکتور، تابع ()act آن فراخوانی می‌شود و تا زمانی که اجرای این تابع به اتمام نرسد، اکتور به طور همروند در حال اجرا باقی می‌ماند. در صورتی که بخواهیم اکتور به طور دائمی در حال اجرا بماند دو راه وجود دارد. راه اول این است که تابع ()act را در پایان کار خود مجدداً فراخوانی کنیم. و راه دیگر استفاده از عبارت **loop** در اسکالا است. دستورات درون حلقه‌ی loop به صورت بی‌پایان اجرا می‌شوند. شکل ۴.۲ کدهای مربوط به این ۲ روش را نمایش می‌دهد.

۲.۲.۲.۲ تبادل پیغام

عملگر ! برای فرستادن پیغام ناهمگام استفاده می‌شود. دستور dest ! message پیغام message را برای اکتور dest ارسال می‌کند بدون آنکه برای دریافت جواب منتظر بماند. با اینکه در مدل اکتور دستوری برای تبادل همگام پیغام وجود ندارد، در اکثر پیاده‌سازی‌ها این امکان به مدل اضافه شده است[۸]. در کتابخانه‌ی اکتور اسکالا، عملگر ?! به این منظور به کار گرفته می‌شود. در صورت استفاده از این دستور، فرستنده‌ی پیغام بلافاصله بعد از ارسال پیغام، تا گرفتن پاسخ متوقف می‌ماند. عملگر دیگری که برای تبادل پیغام مورد استفاده قرار می‌گیرد !! است. این عملگر که در کتابخانه‌ی

```scala
object SillyActor extends Actor {
  def act() {
    for (i <- 1 to 5) {
      println("I'm acting!")
      Thread.sleep(1000)
    }
    act()
  }
}
```

(الف)

```scala
object SillyActor extends Actor {
  def act() {
        loop {
              for (i <- 1 to 5) {
              println("I'm acting!")
              Thread.sleep(1000)
        }
        }
  }
}
```

(ب)

شکل ۴.۲: تداوم اجرای اکتور با استفاده از الف)فراخوانی بازگشتی و ب)حلقه‌ی loop

اسکالا به عنوان آینده[۳۳] شناخته می‌شود، برای حالاتی به کار می‌رود که دریافت پاسخ را می‌توان به صورت محدود به آینده مؤکول کرد. خروجی این عملگر یک تابع است. با فراخوانی این تابع، اکتور تا دریافت پاسخ متناظر متوقف می‌شود. به این ترتیب می‌توان در زمانی که به پاسخ پیغام دریافت شده احتیاج داریم با فراخوانی تابع مربوطه پیغام را دریافت کنیم. برای برداشتن پیغام از صندوق پیغام‌ها، از دو دستور receive و react استفاده می‌شود (تفاوت این دو دستور در بخش ۳.۲.۲.۲ توضیح داده شده است). شکل ۵.۲ مثالی از نحوه‌ی تبادل پیغام بین اکتوران را نمایش می‌دهد. در این برنامه دو اکتور PingActor و PongActor به تبادل پیغام می‌پردازند. در ابتدا اکتور PingActor که متغیر آن با مقدار ۱۰۰ مقداردهی شده است یک پیغام Ping برای اکتور PongActor می‌فرستد و در ادامه در یک حلقه‌ی loop منتظر پاسخ Pong می‌ماند. اکتور PongActor با گرفتن هر پیغام Ping پاسخ Pong را برای فرستنده ارسال می‌کند. کلمه‌ی کلیدی **sender** در کلاس Actor اشاره‌گری به فرستنده‌ی پیغام در حال پردازش می‌باشد (خط ۶ از کد قسمت (ب) شکل ۵.۲). اکتور PingActor با دریافت پاسخ Pong مقدار متغیر pingsLeft را چک می‌کند و در صورت مثبت بودن آن پیغام Ping بعدی را ارسال می‌کند و در غیر این صورت پیغام Stop را ارسال می‌کند. نهایتا با صفر شدن متغیر pingsLeft، اکتور PingActor پیغام Stop را برای PongActor می‌فرستد. دستور exit که در پایان کار هر دو اکتور استفاده شده است باعث می‌شود ریسمان اجرایی اکتور رها شود و پس از اجرای این دستور اکتور قادر به دریافت پیغام نخواهد بود.

[۳۳] Future

```
1 class PingActor(count: int, pong: Actor) extends Actor {
2   def act() {
3     var pingsLeft = count - 1
4     pong ! Ping
5     loop {
6       receive {
7         case Pong =>
8           if (pingsLeft > 0) {
9             pong ! Ping
10            pingsLeft -= 1
11          } else {
12            pong ! Stop
13            exit()
14          }
15      }
16    }
17  }
18 }
```

(الف) اکتور Ping که فرستنده‌ی اولیه‌ی پیغام است

```
1 class PongActor extends Actor {
2   def act() {
3     loop {
4       receive {
5         case Ping =>
6           sender ! Pong
7         case Stop =>
8           Console.println("Pong: stop")
9           exit()
10      }
11    }
12  }
13 }
```

(ب) اکتور Pong که به پیغام ping پاسخ می‌دهد.

```
1 object pingpong extends Application {
2   val pong = new PongActor
3   val ping = new PingActor(100, pong)
4   ping.start
5   pong.start
6 }
```

(ج) کد اجرای برنامه‌ی PingPong

شکل ۲.۵: مثالی از نحوه‌ی تبادل پیغام بین اکتورها

۳.۲.۲.۲ زیرساخت اجرای همروند در کتابخانه‌ی اکتور اسکالا

پردازش‌های همروند مانند اکتورها با دو نوع استراتژی پیاده‌سازی می‌شوند:

- **پیاده‌سازی ریسمان-بنیان:** در این نوع پیاده‌سازی رفتار پردازش همروند به وسیله‌ی یک ریسمان کنترل می‌شود. حالت اجرا[۳۴] به وسیله‌ی پشته‌ی ریسمان[۲۱]
- **پیاده‌سازی رویداد-بنیان:** در این مدل رفتار به کمک یک سری مجری رویداد[۳۵] پیاده‌سازی می‌شوند. این مجری‌ها از یک حلقه‌ی رویداد فراخوانی می‌شوند. حالت اجرای پردازش‌های همروند در این روش به کمک رکوردها یا اشیاء مشخصی که به همین منظور طراحی شده‌اند نگهداری می‌شود [۲۲].

مدل ریسمان-بنیان معمولا پیاده‌سازی راحتتری دارد ولی به دلیل مصرف حافظه‌ی بالا و پرهزینه بودن تعویض متن[۳۶] می‌تواند منجر به کارایی کمتری شود[۲۳]. از طرف دیگر مدل رویداد-بنیان معمولا کاراتر است ولی در طراحی‌های بزرگ پیاده‌سازی آن مشکل‌تر است[۲۴]. استفاده از مدل رویداد-بنیان منجر به ایجاد نوعی از وارونگی کنترل[۳۷] می‌شود: یک برنامه به جای فراخوانی عملیات مسدود کننده [۳۸]، صرفا تمایل خود به ادامه‌ی کار در صورت رخ دادن رویدادهای مشخص (مانند فشردن یک دکمه) را به محیط اجرا اعلام می‌کند. این اعلام تمایل با ثبت یک مجری رویداد در محیط انجام می‌شود. برنامه هیچ وقت این مجری‌های رویداد را فراخوانی نمی‌کند بلکه محیط اجرایی با وقوع هر رخداد، مجری‌های ثبت شده برای آن رویداد را فراخوانی می‌کند. به این ترتیب کنترل اجرای منطق برنامه نسبت به حالت بدون رویداد **وارونه** می‌شود. به دلیل پدیده‌ی وارونگی کنترل، تبدیل یک مدل ریسمان-بنیان به مدل رویداد-بنیان معادل معمولا نیاز به دوباره‌نویسی برنامه دارد[۲۵].

در پیاده‌سازی زیرساخت همروندی در کتابخانه‌ی اکتور اسکالا هر دو رویکرد معرفی شده پیاده‌سازی شده‌اند و قابل دسترسی هستند. اصلی‌ترین عمیات مسدود کننده در مدل اکتور انتظار برای دریافت پیغام است. کنترل اجرا در صورتی مسدود می‌شود که پیغامی که اکتور منتظر دریافت آن است در صندوق پیغام موجود نباشد. در اکتورهای اسکالا، عمل برداشتن پیغام با دو دستور انجام می‌شود:

---

[۳۴] execution state

[۳۵] event handler

[۳۶] context switch

[۳۷] Inversion of Control

[۳۸] blocking operation

- دستور :receive با استفاده از این دستور، در صورتی که در صندوق پیغام اکتور، پیغامی که با یکی از الگوهای معرفی شده در بدنه‌ی receive موجود باشد کد مربوط به الگوی مربوطه اجرا می‌شود. در غیر این صورت ریسمان اجرای این اکتور مسدود می‌شود. در این حالت پشته‌ی فراخوانی تابع ()act در اکتور به صورت خودکار توسط محیط اجرایی ذخیره می‌شود و در صورت ورود پیغام متناسب اجرا به صورت ترتیبی از سر گرفته می‌شود. بنابراین در پیاده‌سازی این دستور از رویکرد ریسمان-بنیان استفاده شده است.
- دستور :react با استفاده از این دستور، در صورتی که هیچ پیغام متناسبی در صندوق پیغام وجود نداشته باشد، به جای مسدود کردن ریسمان اجرای اکتور، از رویکرد رویداد-بنیان استفاده می‌شود. این کار از طریق نوع خاصی از تابع در زبان اسکالا انجام می‌شود که هیچ‌گاه به طور معمولی اجرای آن خاتمه نمی‌یابد. بلکه پس از ثبت مجری رویداد مناسب در محیط اجرا، با استفاده از ایجاد یک استثناء[۳۹] اجرای تابع react و توابع شامل آن در اکتور خاتمه می‌یابد. در این نوع توقف اجرا با توجه به اینکه ریسمان اجرا مسدود نمی‌شود، پشته‌ی فراخوانی تابع نیز ذخیره نمی‌شود و با برگشت به اجرای این تابع، محیط هیچ تاریخچه‌ای از اجرای قبلی آن ندارد. در نتیجه در هر بار بازگشت مانند اولین اجرا رفتار می‌کند. نتیجه‌ی مهم این خصوصیت این است که در صورت استفاده از react در یک اکتور، هیچ کدی که بعد از این تابع نوشته شده باشد اجرا نخواهد شد. به همین دلیل برنامه‌نویس باید دقت کند که تابع react از نظر ترتیب اجرا همیشه آخرین کد بدنه‌ی یک اکتور باشد. نتیجه‌ی استفاده از رویکرد رویداد-بنیان در اکتورهای اسکالا افزایش چشمگیر کارایی در صورت استفاده از تعداد بسیار زیاد اکتور در سیستم است.

به برنامه نویسان توصیه شده است که به جز در موارد خاص که نیاز به مسدود کردن ریسمان اجرای اکتور وجود دارد، در بقیه‌ی موارد از رویکرد رویداد-بنیان استفاده کنند. توضیحات تکمیلی در مورد نحوه‌ی پیاده‌سازی هر دو رویکرد در کتابخانه‌ی اکتور اسکالا و آنالیز کارایی و مقایسه با سایر پیاده‌سازی‌های مدل اکتور در [۲۶] قابل دسترس می‌باشد.

[۳۹] exception

فصل ۳

کارهای پیشین

در این فصل به ارائه‌ی برخی کارهای پیشین و مرتبط به موضوع این پژوهش خواهیم پرداخت. در مورد هر یک از این موارد به ارتباط آن با بحث جاری، کاربرد و یا نقاط تأثیرگذار آن در موضوع این پژوهش و هم‌چنین ضعف ها و نقایص آن‌ها پرداخته شده است.

۱.۳ الگوهای برنامه‌نویسی اکتور

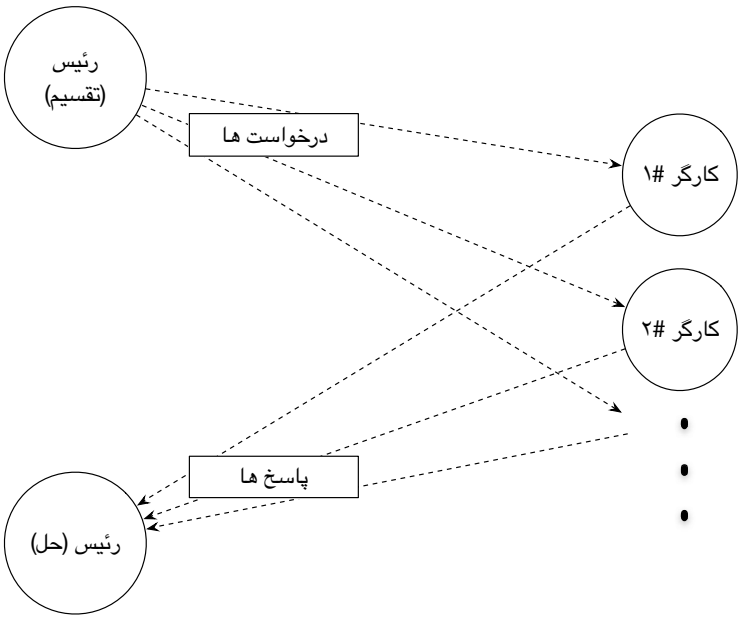
در برنامه‌نویسی همروند با اکتورها دو نوع الگوی کلی معرفی شده است [۶]: یکی *تقسیم-و-حل*[۱] و دیگری *خط لوله*[۲]. در روش تقسیم-و-حل مسئله‌ی مورد بحث به زیربخش‌های کوچکتر و مستقل تقسیم می‌شود که هرکدام به صورت مستقل حل می‌شوند و نتایج هر زیربخش برای نتیجه‌گیری کلی ادغام می‌شوند. در برنامه‌نویسی به مدل اکتور، برای پیاده‌سازی این الگو یک اکتور رئیس[۳] در نظر گرفته می‌شود که تعدادی اکتور کارگر[۴] را برای حل زیربخش‌های مسئله ایجاد می‌کند. عمل تقسیم به وسیله‌ی فرستادن پیغام حاوی حالت لازم برای حل زیر بخش به کارگرها انجام می‌شود.

[۱] devide and conquer

[۲] pipeline

[۳] master

[۴] worker

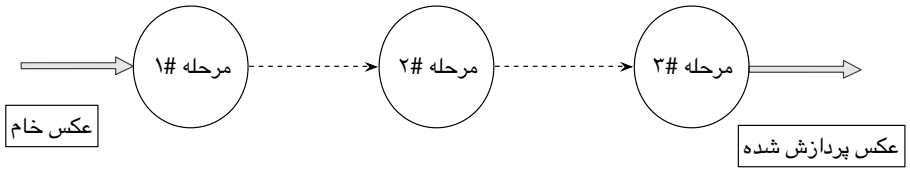


شکل ۱.۳: شمای کلی از الگوی تقسیم-و-حل در مدل اکتور

کارگرها به نوبه‌ی خود منطق لازم برای حل زیر بخش را ایجاد نموده و نتیجه را به صورت پیغام دیگری برای اکتور رئیس ارسال می‌کنند. نهایتا رئیس با ادغام نتایج جواب نهایی مسئله را تولید می‌کند. شایان ذکر است که فازهای تقسیم و حل لزوما توسط اکتور یکسان اجرا نمی‌شوند. ممکن است اجرای فاز حل به اکتور دیگری سپرده شود.[۲۷] مثال دیگری از پیاده‌سازی الگوی تقسیم-و-حل در مدل اکتور در [۲۷] آمده است که در آن الگوریتم جستجوی سریع[۵] توسط این الگو پیاده شده است. شکل ۱.۳ شمایی از نحوه‌ی پیاده‌سازی الگوی تقسیم-و-حل در مدل اکتور را نمایش می‌دهد.
الگوی خط لوله برای حالت‌هایی مناسب است که فعالیت قابل تقسیم به بخش‌های افزایشی باشد. در این صورت هر اکتور تغییرات مربوطه را در مدل ایجاد می‌کند و آن را به عنوان پیغام به اکتور بعدی در خط لوله منتقل می‌کند.

به عنوان مثالی از الگوی خط لوله یک برنامه‌ی پردازش تصویر را در نظر بگیرید. هر مرحله از خط لوله، تغییراتی را در تصویر دریافتی ایجاد می‌کند و تصویر نتیجه را به مرحله‌ی بعد منتقل می‌کند. در پیاده‌سازی با روش اکتور، هر مرحله به صورت یک اکتور مدل می‌شود و تصویر به صورت پیغام بین مراحل رد و بدل می‌شود. در شکل ۲.۳ شمایی از این الگو نشان داده شده است.
در پژوهش‌های انجام شده مشخص شد که الگوهای ارائه شده صرفا الگوهای کلی همروندی هستند و جزئیات این الگوها در طراحی منطق دامنه، نحوه‌ی طراحی پیغام‌ها بررسی نشده اند .

[۵] quick sort



شکل ۲.۳: مثالی از الگوی خط لوله (پردازش تصویر)

۲.۳ همگام‌سازی و هماهنگی اکتورها

همان‌طور که در بخش‌های قبل ذکر شد، مدل اکتور دارای خاصیت ناهمگامی است و ترتیب پیغام‌هایی که یک اکتور دریافت می‌کند وابسته به ترتیب فرستاده شدن پیغام‌ها نیست. نتیجه‌ی این خاصیت این است که تعداد ترتیب[۶]های دریافت پیغام‌ها در مدل اکتور نمایی است[۷]. به دلیل اینکه فرستنده‌ی پیغام از حالت محلی اکتور گیرنده اطلاعی ندارد، ممکن است بعضی از ترتیب‌های ذکر شده برای پیغام‌ها مطلوب نباشد. به عنوان مثال الگوریتمی را در نظر بگیرید که زیر بخش‌های مختلف آن به اکتورهایی فرستاده شده و نتایج آن دریافت می‌شود ولی در آن ترتیب دریافت نتایج اهمیت داشته باشد. نیاز به این نوع اولویت‌بندی‌ها در مدل اکتور منجر به ایجاد پیچیدگی در محاسبات همروند می‌شود و در صورت پیاده‌سازی نامناسب باعث ایجاد ناکارامدی در برنامه‌ها می‌شود. راه حل این مسئله در مدل اکتور همگام‌سازی است. در مدل اکتور، اکتورها برای همگام‌سازی باهم ارتباط برقرار می‌کنند. در این قسمت دو نوع الگوی هماهنگی اکتورها را معرفی می‌کنیم: تبادل پیغام شبه آرپی‌سی (فراخوانی رویه راه دور)[۷] و قیود همگام‌سازی محلی[۸]. [۶، ۲۸، ۲۹، ۷]

[۶] ordering

[۷] Remote Procedure Call

[۸] Local Synchronization Constraints

۱.۲.۳ تبادل پیغام شبه-آرپی‌سی

در ارتباط شبه-آرپی‌سی، فرستنده پس از ارسال پیغام منتظر گرفتن پیغام پاسخ از طرف گیرنده می‌ماند. رفتار اکتور در این مدل به ترتیب زیر است:

۱. اکتور فرستنده درخواست را در قالب یک پیغام به اکتور گیرنده ارسال می‌کند.

۲. سپس فرستنده صندوق پیغام‌ها را بررسی می‌کند.

۳. اگر پیغام بعدی پاسخ درخواست ارسال شده باشد اقدام مناسب صورت می‌گیرد و فعالیت اکتور ادامه پیدا می‌کند.

۴. اگر پیغام بعدی پاسخ درخواست ارسال شده نباشد پیغام جاری در صورت امکان (بسته به منطق برنامه) پردازش می‌شود و در غیر این صورت برای پردازش در آینده به صندوق پیغام‌ها برگردانده می‌شود.

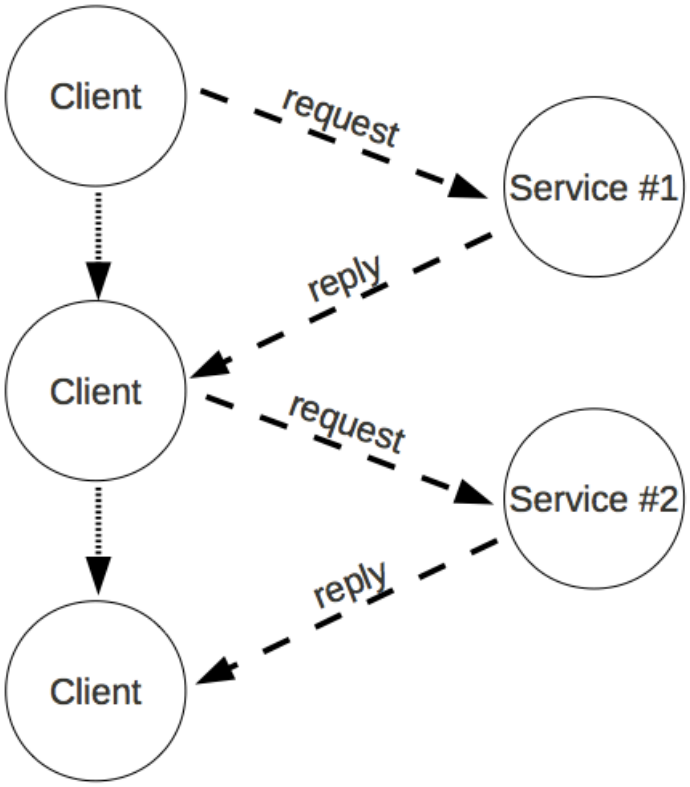
شکل ۳.۳ مثالی از پیاده‌سازی ارتباط شبه-آرپی‌سی در مدل اکتور را نشان می‌دهد. ارتباط شبه-آرسی‌پی در دو نوع سناریوی خاص مفید و ضروری است: یک سناریو این است که اکتور نیاز به ارسال پیغام به صورت ترتیبی به یک یا چند اکتور خاص دارد و تا حاصل شدن اطمینان از رسیدن پیغام قبلی پیغام بعد را ارسال نمی‌کند. سناریوی دوم این است که حالت[۹] اکتور فرستنده بستگی به محتوای پاسخ دارد. در این حالت اکتور قبل از دریافت پاسخ مورد نظر، نمی‌تواند پیغام‌های بعدی را به درستی پردازش کند. نکته‌ی قابل توجه این است که با توجه به شباهت ارسال پیغام شبه-آرپی‌سی به فراخوانی رویه[۱۰]ها در زبان‌های ترتیبی[۱۱]، معمولا برنامه‌نویسان گرایش به استفاده‌ی بیش از حد از این نوع تبادل پیغام دارند که این ممکن است با ایجاد وابستگی‌های بی‌مورد در اشیاء برنامه، علاوه بر کاهش کارایی، منجر به ایجاد بن‌باز[۱۲] در برنامه شود (حالتی که یک اکتور به علت انتظار برای پاسخی که هرگز دریافت نخواهد کرد، از پیغام‌های جدید مرتباً چشم‌پوشی می‌کند یا پردازش آنها را به تأخیر می‌اندازد).
امکان تبادل پیغام شبه-آرپی‌سی تقریبا در تمامی پیاده‌سازی‌های مدل اکتور به صورت امکانات سطح زبان وجود دارد[۸].

[۹] state

[۱۰] procedure

[۱۱] sequential

[۱۲] live lock



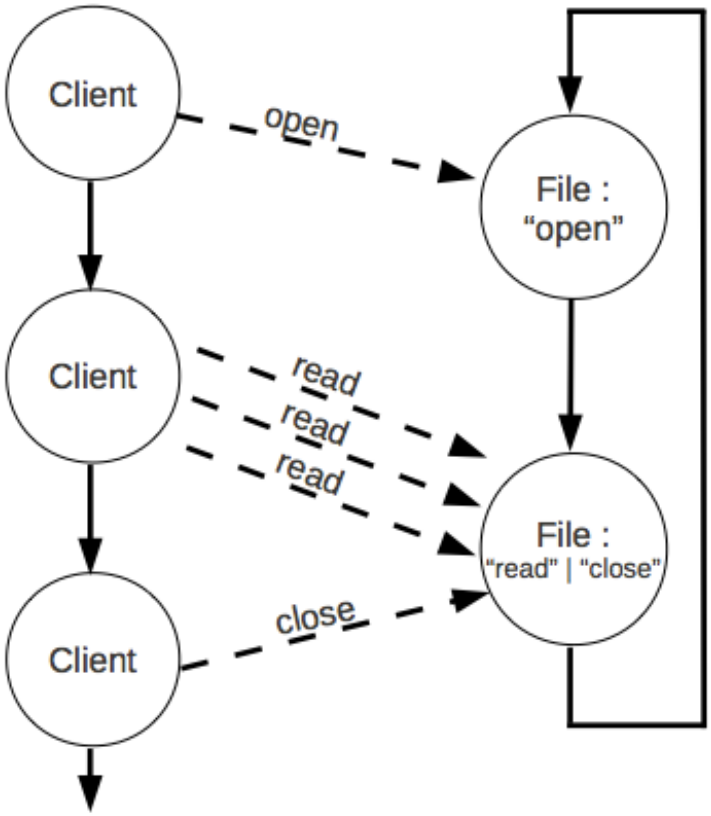
شکل ۳.۳: مثالی از ارتباط شبه-آرپی‌سی در اکتورها)

۲.۲.۳ قیود همگام‌سازی محلی

استفاده از قیود همگام‌سازی محلی روشی برای اولیت‌بندی پردازش پیغام‌ها در مدل اکتور است[۳۰]. برای توضیح مفهوم همگام‌سازی محلی مثالی در شکل ۴.۳ ارائه شده است. در این مثال اکتور فایل پس از دریافت پیغام باز کردن فایل[۱۳]، با استفاده از قیود همگام‌سازی خود را محدود به پردازش پیغام‌های *بستن*، *خواندن* می‌کند. در صورت عدم وجود امکانات مناسب برای قیود همگام‌سازی، برنامه‌نویس ناگزیر خواهد بود تا در میان منطق اجرای پیغام‌ها، میانگیر صندوق پیغام‌ها را بررسی و ترکیب یا ترتیب آنها را تغییر داده و یا با جستجو در آنها پیغام مناسب را انتخاب کند. این امر موجب مخلوط شدن منطق چگونگی پردازش پیغام (چگونه) با منطق زمانی انتخاب پیغام (چه زمانی) می‌شود که در اصول نرم‌افزار پدیده‌ی نامطلوبی به حساب می‌آید[۷]. به همین دلیل بسیاری از زبان‌ها و چارچوب‌های مبتنی بر اکتور امکانات مناسبی برای پشتیبانی از قیود همگام‌سازی محلی ارائه داده‌اند. به عنوان مثال در کتابخانه‌ی اکتور اسکالا که در بخش ۲.۲.۲ معرفی شد، از مکانیزم تطابق الگو[۱۴] برای اولیت بندی پردازش پیغام‌ها بدون اینکه با منطق اجرایی برنامه مخلوط گردد استفاده می‌شود.

[۱۳] open

[۱۴] pattern matching



شکل ۴.۳: مثالی از قیود همگام‌سازی محلی. اکتور فایل به وسیله‌ی قیود همگام‌سازی محدود شده است. فلش عمودی به معنی ترتیب زمانی و برچسب‌های داخل دایره به معنی پیغام‌های قابل پردازش در هر حالت هستند.)

فصل ۴

طراحی بر اساس تبادل ناهمگام پیغام

۱.۴ مقدمه

در این فصل از پژوهش روش طراحی منطق دامنه بر اساس تبادل ناهمگام پیغام ارائه شده است. تلاش شده است تا تطابق طراحی با مدل بازیگر در حد امکان حفظ شود. با توجه به تمرکز این بخش بر روش طراحی منطق دامنه و به هدف ایجاد شفافیت و افزایش قابلیت فهم نکات و الگوهای مطرح شده در روش، تصمیم به استفاده از یک سیستم نمونه به عنوان مثال گرفته شد. کلیه‌ی نکات مطرح شده در ادامه‌ی این بخش در قالب این مثال ارائه خواهند شد. در انتخاب سیستم نمونه نکات ذیل مورد توجه قرار گرفته است:

۱. **دامنه‌ی سیستم انتخابی:** رده‌ی دامنه‌ی سیستم انتخاب شده به طور کلی سیستم‌های اطلاعاتی[۱] است. اولین دلیل انتخاب این رده این است که در این نوع دامنه همروندی به طور ذاتی وجود ندارد و به همین دلیل زمینه‌ی مقایسه‌ی طراحی بر اساس تبادل ناهمگام با طراحی‌های شیءگرای ترتیبی فراهم می‌شود. با توجه به اینکه یکی از موارد مقایسه‌ی این نوع طراحی با طراحی شیءگرای ترتیبی تفاوت کارایی این دو رویکرد است، دامنه‌ی انتخاب شده باید در حالت ترتیبی هم قابلیت اضافه شدن همروندی را داشته باشد. سیستم‌های اطلاعاتی از این حیث نیز انتخاب مناسبی محسوب می‌شوند چرا که در اکثر پیاده‌سازی‌های عملیاتی، علیرغم داشتن طراحی ترتیبی، به

[۱] Information System

وسیله‌ی ریسمان‌هایی که وب‌سرورها برای پاسخگویی به درخواست‌های همزمان کاربران ایجاد می‌کنند، دارای خاصیت همروندی نیز می‌گردند. به همین دلیل در بخش ارزیابی می‌توانیم با شبیه‌سازی عملیات وب‌سرورها، کارایی و نیز تغییرپذیری دو نوع طراحی مذکور را ارزیابی و مقایسه کنیم. دلیل دیگر این انتخاب بالا بودن میزان آشنایی جامعه‌ی طراحی شیءگرا با این نوع سیستم‌ها و استفاده‌ی گسترده از این نوع سیستم‌ها می‌باشد. شایان ذکر است که سعی شده است در ارائه‌ی الگوها و نکات استخراج شده از این طراحی بر دامنه‌ی انتخاب شده تکیه نشود. دامنه‌ی سیستم نمونه نیز یک سیستم آموزشی انتخاب شده است. با توجه به اینکه استفاده کنندگان این پژوهش جامعه‌ی دانشگاهی هستند، آشنایی این جامعه با سیستم آموزشی دلیل اصلی انتخاب آن بوده است.

۲. **بزرگی منطق دامنه:** از نظر میزان بزرگی سیستم (تعداد کلاس‌ها و موارد کاربرد[۲])، سعی شده منطق حداقل بزرگی و پیچیدگی را داشته باشد تا ضمن امکان مشاهده‌ی الگوهای مختلف، نیازی به تکرار نکات طراحی برای مولفه‌های متعدد و مشابه نباشد.

۲.۴ معرفی یک سیستم آموزش ساده

همان‌طور که در بخش ۱.۴ ذکر شد، یک سیستم آموزش کوچک به عنوان مدل طراحی انتخاب شده است. در ادامه‌ی این بخش ابتدا موارد کاربرد[۳] انتخاب شده در این سیستم را توصیف می‌کنیم و سپس با توجه به آنها مدل دامنه[۴] سیستم را در قالب نمودار کلاس بیان می‌کنیم.

۱.۲.۴ موارد کاربرد

در این بخش موارد کاربرد انتخاب شده برای سیستم آموزش معرفی می‌شوند. لازم به تأکید است که علیرغم این که این موارد کاربرد، مرتبط و هماهنگ با موارد کاربرد یک سیستم آموزش واقعی هستند، به هیچ عنوان تمام موارد کاربرد مورد نیاز برای ساختن سیستم واقعی را شامل نمی‌شوند و علاوه بر آن، موارد انتخاب شده دارای جزئیات و دقت کافی برای پوشش فرایندهای واقعی نیستند. در ادامه‌ی این بخش، هر مورد کاربرد در قالب یک جدول توصیفی ارائه شده است.

[۲] use cases

[۳] use cases

[۴] Domain Model

نام مورد کاربرد	درخواست محاسبه‌ی معدل ترم دانشجو
بازیگر(ان)	کاربر
شروع می‌شود زمانی که	درخواست محاسبه‌ی معدل ترم وارد سیستم می‌شود.
پیش‌شرط‌ها	دانشجو و ترم در سیستم تعریف شده باشند.
جریان اصلی	۱. درخواست محاسبه‌ی معدل دانشجو در ترم مربوطه وارد سیستم می‌شود. ۲. سیستم سوابق تحصیلی دانشجو در ترم مربوطه را بررسی می‌کند. معدل ترم با توجه به نمرات اخذ شده و تعداد واحد هر درس محاسبه و اعلام می‌شود. در صورتی که نمره‌ی درس سابقه‌ای وارد نشده باشد، درس مربوطه در محاسبه‌ی معدل لحاظ نمی‌گردد.
جریان استثنا ۱	۲.الف) در صورتی که دانشجو هیچ واحدی در ترم جاری اخذ نکرده باشد پیغام خطای مناسب صادر می‌شود و جریان اصلی خاتمه می‌یابد.
تمام می‌شود زمانی که	معدل دانشجو اعلام می‌شود یا خطای مناسب صادر می‌گردد.

جدول ۱.۴: توصیف مورد کاربرد محاسبه‌ی معدل یک دانشجو در یک ترم

نام مورد کاربرد	درخواست اخذ یک ارائه در یک ترم
بازیگر(ان)	کاربر
شروع می‌شود زمانی که	درخواست اخذ ارائه وارد سیستم می‌شود.
پیش‌شرط‌ها	۱. انتخاب واحد در ترم امکانپذیر باشد. (رجوع کنید به جداول ۴.۴ و۳.۴)
جریان اصلی	۱. سیستم کنترل می‌کند که دانشجو در ترم‌های قبل این درس را نگذرانده باشد. ۲. سیستم کنترل می‌کند که دانشجو در ترم جاری این درس را اخذ نکرده باشد. ۳. سیستم کنترل می‌کند که دانشجو تمام پیش‌نیازهای این درس را با موفقیت گذرانده باشد. ۴. سیستم کنترل می‌کند که تعداد واحدهای اخذ شده توسط دانشجو در این ترم پس از اخذ این درس بیشتر از ۲۰ نشود. ۵. سیستم یک سابقه از ارائه‌ی انتخاب شده برای دانشجو تشکیل می‌دهد و آن را در سوابق دانشجو ثبت می‌کند.
جریان استثنا ۱	۱.الف)در صورتی که دانشجو قبلا این درس را گذرانده باشد، خطای "درس انتخاب شده قبلاً گذرانده شده است" صادر می‌شود و جریان اصلی خاتمه می‌یابد.
جریان استثنا ۲	۲.الف)در صورتی که دانشجو در ترم جاری این درس را اخذ کرده باشد، خطای "این درس در ترم جاری قبلاً اخذ شده است" صادر می‌شود و جریان اصلی خاتمه می‌یابد.
جریان استثنا ۳	۳.الف)در صورتی که دانشجو یکی از پیش‌نیازهای درس را نگذرانده باشد، خطای "انتخاب بیشتر از ۲۰ واحد در ترم مجاز نمی‌باشد" صادر می‌شود و جریان اصلی خاتمه می‌یابد.
جریان استثنا ۴	۴.الف)در صورتی که تعداد واحدهای اخذ شده توسط دانشجو در این ترم پس از اخذ این درس بیشتر از ۲۰ شود، خطای "انتخاب بیشتر از ۲۰ واحد در ترم مجاز نمی‌باشد" صادر می‌شود و جریان اصلی خاتمه می‌یابد.
تمام می‌شود زمانی که	سابقه‌ی جدید در سوابق دانشجو ثبت می‌شود و یا خطای مناسب صادر می‌گردد.

جدول ۲.۴: توصیف مورد کاربرد اخذ یک ارائه توسط یک دانشجو در یک ترم

نام مورد کاربرد	درخواست غیر فعال کردن ارائه‌های یک ترم برای انتخاب واحد
بازیگر(ان)	کاربر(مدیر سیستم)
شروع می‌شود زمانی که	درخواست غیرفعال کردن ارائه‌های یک ترم وارد سیستم می‌شود.
پیش‌شرط‌ها	ترم در سیستم تعریف شده باشند.
جریان اصلی	۱. درخواست غیر فعال کردن ارائه‌های یک ترم وارد سیستم می‌شود. ۲.سیستم تمام ارائه‌های یک ترم را غیرفعال می‌کند.
تمام می‌شود زمانی که	تمام ارائه‌های ترم برای انتخاب واحد غیرفعال می‌شوند.
پس شرط‌ها	انتخاب واحد در ترم امکان پذیر نیست.

جدول ۳.۴: توصیف مورد کاربرد غیرفعال کردن ارائه‌های یک ترم برای انتخاب واحد

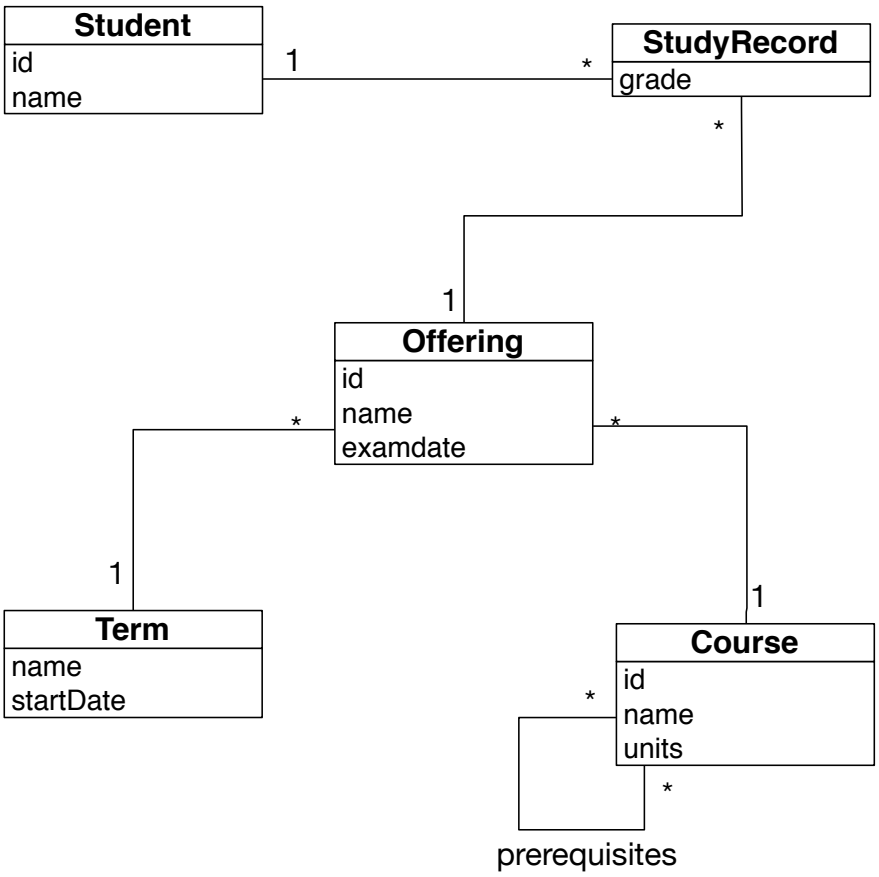
نام مورد کاربرد	درخواست فعال کردن ارائه‌های یک ترم برای انتخاب واحد
بازیگر(ان)	کاربر(مدیر سیستم)
شروع می‌شود زمانی که	درخواست فعال کردن ارائه‌های یک ترم وارد سیستم می‌شود.
پیش‌شرط‌ها	ترم در سیستم تعریف شده باشند.
جریان اصلی	۱. درخواست فعال کردن ارائه‌های یک ترم وارد سیستم می‌شود. ۲.سیستم تمام ارائه‌های یک ترم را فعال می‌کند.
تمام می‌شود زمانی که	تمام ارائه‌های ترم برای انتخاب واحدفعال می‌شوند.
پس شرط‌ها	انتخاب واحد در ترم امکان پذیر است.

جدول ۴.۴: توصیف مورد کاربرد فعال کردن ارائه‌های یک ترم برای انتخاب واحد

۲.۲.۴ اشیاء دامنه

موجودیت‌های اصلی مدل ابتدایی این سیستم عبارتند از: ***دانشجو***[۵]، ***درس***[۶]، ***ترم***[۷]، ***ارائه***[۸] و ***سابقه***[۹].
در هر *ترم* تحصیلی، تعدادی *ارائه* از دروس مختلف وجود دارد. هر درس می‌تواند *ارائه*‌های مختلفی داشته باشد. به عنوان مثال درس ریاضی۱ می‌تواند در ترم ۱-۹۱-۹۰سه ارائه‌ی مختلف داشته باشد. دانشجو با اخذ هر ارائه *سابقه*‌ای از آن ارائه را به اسم خود ثبت می‌کند. در این سابقه اطلاعاتی مثل نمره‌ی دانشجو و وضعیت قبول یا مردودی درس در طول ترم ثبت خواهد شد. دروس می‌توانند رابطه‌ی پیش‌نیازی[۱۰] باهم داشته باشند. شکل ۱.۴ مدل دامنه‌ی سیستم را به وسیله‌ی یک نمودار کلاس مبتنی بر **یوام‌ال**[۱۱] نشان می‌دهد.

[۵] Student
[۶] Course
[۷] Term
[۸] Offering
[۹] Study Record
[۱۰] prerequisite
[۱۱] UML



شکل ۱.۴: نمودار کلاس مدل ابتدای سیستم آموزش ساده

۳.۴ طراحی سیستم آموزش به روش تبادل ناهمگام پیغام

در این بخش طراحی سیستم معرفی شده در بخش ۲.۴ به روش تبادل ناهمگام پیغام ارائه می‌گردد. سعی شده است تا به جای ارائه‌ی یکباره‌ی طراحی نهایی، یک رویکرد افزایشی[۱۲] برای طراحی اتخاذ شود. در این رویکرد مراحل تشکیل نهایی طرح و حتی اقدامات اشتباهی که در طول طراحی برداشته شده است ارائه خواهد شد. به این ترتیب علاوه بر قابل استفاده‌تر بودن پژوهش به صورت یک دستورالعمل [۱۳] طراحی، قابلیت فهم روش طراحی هم بالاتر می‌رود.

۱.۳.۴ طراحی اکتورهای مدل دامنه

اکتورهای اصلی سیستم همان اشیاء مدل دامنه هستند که در بخش ۲.۲.۴ معرفی شدند.البته احتمالاً علاوه بر این اکتورها، اکتورهای دیگری نیز برای پیاده‌سازی کارکردهای سیستم استفاده خواهند شد. در طراحی اکتورهای اصلی صرفا فیلدهای داده‌ای اکتور و نیز پیغام‌های اصلی که از روابط موجود در نمودار کلاس ۱.۴ قابل استخراج هستند در نظر گرفته می‌شود. منطق پیاده‌سازی عملیات هر پیغام و پیغام‌های دیگری که به این منظور ایجاد می‌شوند در ادامه به طراحی افزوده خواهد شد. با توجه به اینکه در مدل اکتور، تنها راه ارتباط بین اکتورها استفاده از تبادل پیغام است و این که یک اکتور برای امکان ارسال پیغام به اکتور دیگر نیاز به دسترسی به اسم آن دارد، بهترین راه برای طراحی رابطه‌های وابستگی[۱۴] این است که در کلاس یک اکتور برای هر کلاس دیگر که رابطه‌ای با آن وجود دارد یک فیلد از نوع کلاس طرف دیگر در نظر گرفته شود. این مورد مشابه طراحی شیءگرای عادی (ترتیبی) است. از طرف دیگر در مدل طراحی شیءگرای ترتیبی برای هر کارکرد اصلی یک شیء نیز یک متد در کلاس متناظر با آن در نظر گرفته می‌شود که برای اجرای کارکرد، متد مورد نظر فراخوانی می‌شود. با توجه به اینکه در مدل اکتور مکانیزم کنترلی برنامه به جای فراخوانی متد، تبادل پیغام است، باید به ازای هر متد متناظر در حالت شیءگرا، یک پیغام دریافت شود. البته در این مرحله از طراحی منطق پیاده‌سازی کارکرد هر پیغام در نظر گرفته نشده است و در مراحل بعدی به تدریج اضافه خواهد شد.

۱. **اکتور دانشجو:** این اکتور دارای فیلدهای نام و شناسه است. به علت ارتباط دانشجو با سابقه‌ها و نیاز به ارسال پیغام به آنها یک فیلد از نوع لیست سابقه نیز در کلاس دانشجو وجود دارد. قطعه کد ۲.۴ طرح ابتدایی کلاس

[۱۲] incremental

[۱۳] receipe

[۱۴] association

```
1 class Student(
2   var id: String,
3   var name: String,
4   var studyRecords: List[StudyRecord]) extends Actor {
5
6   override def act() {
7     loop {
8       react {
9         case HasPassed(course, target) =>
10          ...
11        case HasTaken(course, target) =>
12          ...
13        case GPARequest(term: Term, target: Actor) =>
14          ...
15        case TakeCourse(offering, target) =>
16      }
17    }
18  }
19 }
```

شکل ۲.۴: ساختار کلاس اکتور دانشجو

اکتور دانشجو را نشان می‌دهد. همان‌طور که در بخش قبل ذکر شد منطق پیاده‌سازی کارکرد پیغام‌ها در این مرحله اضافه نشده و در ادامه‌ی فصل به تدریج تکمیل خواهد شد. پیغام‌هایی که اکتور دانشجو دریافت می‌کند عبارتند از:

(آ) **GPARequest(term):** با دریافت این پیغام دانشجو باید پاسخ دهد که معدل دانشجو در ترم جاری چند بوده است.

(ب) **TakeCourse(offering):** با دریافت این پیغام دانشجو باید درس ارائه‌ی مربوطه را اخذ کند. طبیعتاً تمام شرایط ذکر شده در مورد کاربرد ۲.۴ باید بررسی شود.

طبیعتاً این موارد تنها شامل پیغام‌هایی است که مستقیماً از موارد کاربرد قابل استخراج هستند. در هنگام طراحی کارکردهای موارد کاربرد، در صورت لزوم پیغام‌های جدیدی به این کلاس اضافه خواهد شد.

۲. **اکتور سابقه:** مطابق مدل دامنه که در شکل ۱.۴ ارائه شده است، تنها فیلد داده‌ای این اکتور، نمره است. به علت

```
1 class StudyRecord(
2   var grade: Double,
3   var offering: Offering) extends Actor {
4   def act() {
5     loop {
6       react {
7         case ...
8       }
9     }
10  }
```

شکل ۳.۴: ساختار کلاس اکتور سابقه

ارتباط سابقه با اکتور ارائه، یک فیلد از نوع ارائه نیز در کلاس سابقه وجود دارد. قطعه کد ۳.۴ طرح ابتدایی کلاس اکتور سابقه را نشان می‌دهد. همان‌طور که در بخش قبل ذکر شد منطق پیاده‌سازی کارکرد پیغام‌ها در این مرحله اضافه نشده و در ادامه‌ی فصل به تدریج تکمیل خواهد شد. در این مرحله، پیغامی که مستقیماً از موارد کاربرد قابل استخراج باشد وجود ندارد و در هنگام طراحی کارکردهای موارد کاربرد، در صورت لزوم پیغام‌های جدیدی به این کلاس اضافه خواهد شد.

۳. **اکتور ارائه:** مطابق مدل دامنه که در شکل ۱.۴ ارائه شده است، فیلدهای داده‌ای این اکتور عبارتند از شناسه و تاریخ امتحان[۱۵]. به علت ارتباط ارائه با اکتورهای درس و ترم، یک فیلد از نوع درس و یک فیلد از نوع ترم نیز در کلاس ارائه وجود دارد. قطعه کد ۴.۴ طرح ابتدایی کلاس اکتور ارائه را نشان می‌دهد. در این مرحله، پیغامی که مستقیماً از موارد کاربرد قابل استخراج باشد وجود ندارد و در هنگام طراحی کارکردهای موارد کاربرد، در صورت لزوم پیغام‌های جدیدی به این کلاس اضافه خواهد شد.

۴. **اکتور درس:** مطابق مدل دامنه که در شکل ۱.۴ ارائه شده است، فیلدهای داده‌ای این اکتور عبارتند از شناسه، نام و تعداد واحد. تنها ارتباط این کلاس که نیاز به ایجاد فیلد دارد ارتباط دروس پیش‌نیاز است. بنابراین یک فیلد از نوع لیست درس نیز به این منظور باید به کلاس اضافه شود. قطعه کد ۵.۴ طرح ابتدایی کلاس اکتور درس را نشان می‌دهد. در این مرحله، پیغامی که مستقیماً از موارد کاربرد قابل استخراج باشد وجود ندارد و در هنگام طراحی کارکردهای موارد کاربرد، در صورت لزوم پیغام‌های جدیدی به این کلاس اضافه خواهد شد.

[۱۵] examDate

```
class Offering(
  var id: String,
  var examDate: Date,
  var course: Course,
  var term: Term) extends Actor {
  def act() {
    loop {
      react {
        case ...
      }
    }
  }
```

شکل ۴.۴: ساختار کلاس اکتور ارائه

```
class Course(
  var id: String,
  var name: String,
  var units: Int,
  var preRequisites: List[Course]) extends Actor {
  def act() {
    loop {
      react {
        case ...
      }
    }
  }
```

شکل ۵.۴: ساختار کلاس اکتور درس

```
1 class Term(
2   var name: String,
3   var startDate: Date) extends Actor {
4   def act() {
5     loop {
6       react {
7         case ...
8       }
9     }
10  }
```

شکل ۶.۴: ساختار کلاس اکتور ترم

۵. **اکتور ترم:** مطابق مدل دامنه که در شکل ۱.۴ ارائه شده است، فیلدهای داده‌ای این اکتور عبارتند از نام و تاریخ شروعstartDate. با توجه به موارد کاربرد مطرح شده، اکتور ترم آغاز کننده‌ی هیچ ارتباطی نیست و به همین دلیل نیازی به داشتن فیلدی برای این منظور نیست. اکتور ترم قطعه کد ۶.۴ طرح ابتدایی کلاس اکتور ترم را نشان می‌دهد. در این مرحله، پیغامی که مستقیماً از موارد کاربرد قابل استخراج باشد وجود ندارد و در هنگام طراحی کارکردهای موارد کاربرد، در صورت لزوم پیغام‌های جدیدی به این کلاس اضافه خواهد شد.

۲.۳.۴ مورد کاربرد محاسبه‌ی معدل

این مورد کاربرد در جدول ۱.۴ توصیف شده است.

۱.۲.۳.۴ رویکرد اول

برای محاسبه‌ی معدل ترم یک دانشجو نیاز داریم نمره‌ی تمام درس‌های دانشجو در ترم به همراه تعداد واحدهای آن درس‌ها را در اختیار داشته باشیم. درخواست معدل برای ترم از طرف دانشجو صورت می‌گیرد بنابراین شروع پیغام‌ها از این اکتور آغاز می‌شود. اکتور دانشجو به هر کدام از اکتورهای سابقه[۱۶] یک پیغام می‌فرستد و به وسیله‌ی آن اعلام می‌کند نمره و تعداد واحدهای درس مربوط به سابقه در پاسخ ارسال شود. علاوه بر این، در پاسخ باید مشخص شود که آیا سابقه مربوط به همان ترم است که معدل برای آن درخواست شده یا خیر. بنابراین پیغام‌های درخواست نمره برای معدل و پاسخ آن به صورت زیر خواهند بود:

request: GPAInfoRequest(term: Term)

response: GPAInfoResponse(isForTerm:Boolean, grade: Double, units:Int)

اکتور سابقه امکان اینکه بدون برقراری ارتباط با اکتور ارائه[۱۷] جواب این پیغام را بدهد، ندارد. دلیل این امر این است که اولا سابقه لزوما مربوط به ترمی نیست که معدل برای آن درخواست شده است، ثانیا سابقه اطلاعی از تعداد واحدهای درس مربوطه ندارد. به همین دلیل، سابقه باید برای جمع‌آوری این اطلاعات با اکتورهای دیگر تبادل پیغام انجام دهد. از طرف دیگر تنها اکتوری که به نمره‌ی دانشجو دسترسی ادارد، اکتور سابقه است. در نتیجه فرستادن پاسخ به درخواست دانشجو نیاز به همکاری ۳ اکتور سابقه، درس و ترم دارد. با توجه به اینکه دسترسی سابقه به اکتورهای درس و ترم از طریق اکتور ارائه ممکن می‌شود، این اکتور نیز در تبادل پیغام‌ها مشارکت خواهد داشت.
با توجه به موارد ذکر شده، اکتور سابقه دو راهکار پیش رو دارد:

۱. اکتور سابقه به وسیله‌ی درخواست‌هایی، تعیین کند که ترم مربوط به این سابقه همان ترم مورد درخواست در معدل است یا خیر، و نیز تعداد واحدهای درس چند است. و در ادامه با ترکیب این اطلاعات با نمره‌ی سابقه، خود

[۱۶] StudyRecord

[۱۷] Offering

پاسخ اکتور دانشجو را ارسال کند.

۲. اکتور سابقه نمره را در پاسخ قرار دهد ولی با توجه به اینکه پاسخ هنوز کامل نیست (هنوز معلوم نیست که درس چند واحدی است و آیا مربوط به ترم درخواستی است یا خیر)، به جای اینکه پاسخ را برای دانشجو پس بفرستد، آن را برای تکمیل به اکتور ارائه منتقل کند.

در این رویکرد فرض بر انتخاب اول است، یعنی اینکه خود اکتور سابقه، با گرفتن اطلاعات مورد نیاز از ارائه، پاسخ دانشجو را ارسال می‌کند.

برای این کار اکتور سابقه پیغام GPAInfoRequest را برای اکتور ارائه ارسال می‌کند و منتظر دریافت پاسخ می‌ماند. اکتور ارائه با دریافت GPAInfoRequest دو پیغام به صورت زیر به ترتیب برای اکتور ترم و اکتور درس ارسال می‌کند و منتظر پاسخ آنها می‌ماند:

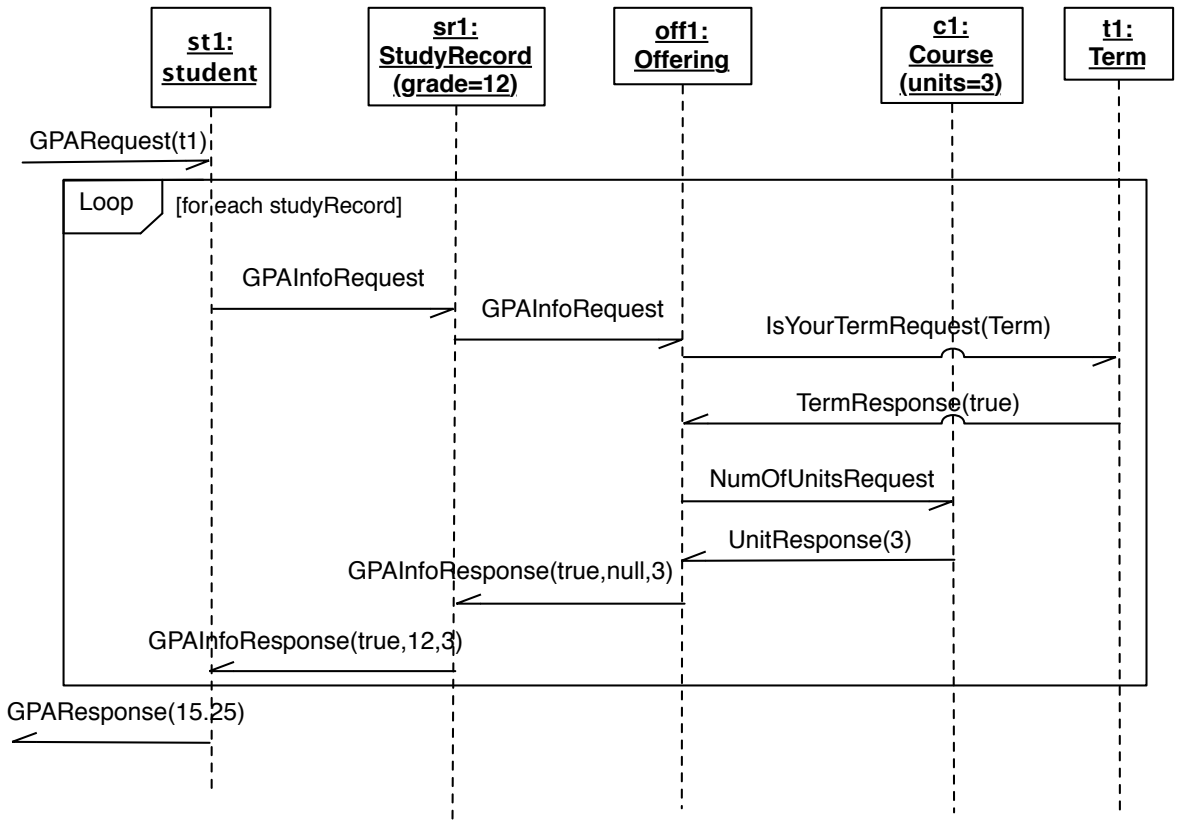
IsYourTermRequest(term: Term)

NumOfUnitsRequest

هدف از درخواست اول این است که مشخص شود که درسی که سابقه به آن متعلق است، متعلق به همان ترمی است که معدل برای آن درخواست شده یا خیر (اگر جواب خیر باشد نمره‌ی درس در معدل در نظر گرفته نخواهد شد). پیغام دوم هم تعداد واحدهای درس را از اکتور درس درخواست می‌کند. ترم و درس به سادگی به این دو پیغام پاسخ می‌دهند و ارائه با گرفتن پاسخ‌ها، اطلاعات آنها را تجمیع[۱۸] کرده و برای اکتور سابقه ارسال می‌کند. سابقه با دریافت این پیغام، به تمام اطلاعات لازم برای این که پاسخ اکتور دانشجو را بدهد، دسترسی دارد. بنابراین می‌تواند با اضافه کردن مقدار فیلد نمره‌ی خود به پیغام آن را برای دانشجو ارسال کند. دانشجو با گرفتن این پاسخ، یکی از نمره‌های لازم برای محاسبه‌ی معدل را در دست دارد. بقیه‌ی نمره‌ها از تکرار همین عملیات برای تمام اکتورهای سابقه‌ی مربوط به دانشجو به طور مشابه به دست می‌آیند. در نهایت اکتور دانشجو با جمع نمراتی که مربوط به ترم درخواستی بوده‌اند (که از مقدار فیلد isForTerm از پیغام‌های پاسخ قابل تشخیص است) و تقسیم آن بر جمع واحدهای مربوط به ترم (فیلد units پیغام پاسخ) معدل را محاسبه کرده و برای اکتوری که درخواست معدل داده ارسال می‌کند. شکل ۷.۴ نمودار ترتیب[۱۹] برای

[۱۸] منظور از تجمیع در اینجا این است که پاسخ فرضی true برای پیغام IsYourTermRequest(term) و پاسخ فرضی ۳ برای پیغام NumOfUnitsRequest را که به ترتیب از اکتورهای ترم و درس گرفته شده، به صورت پیغام GPAInfoResponse(isForTerm=true,grade=null,unit=3) باهم ترکیب می‌کند.

[۱۹] sequence diagram



شکل ۷.۴: نمودار ترتیب برای رویکرد اول محاسبه‌ی معدل

پیغام‌های مبادله شده در این رویکرد را در قالب یک مثال نشان می‌دهد. در بخشی از این مثال که در شکل قابل مشاهده است فرض شده ترم مربوط به درخواست معدل باشد و تعداد واحدهای درس ۳ باشد. نمره‌ی سابقه‌ای که درخواست برای آن ارسال شده ۱۲ است. درنهایت پس از تکرار حلقه‌ی مشخص شده در شکل و ارسال پیغام‌ها به تمام سابقه‌ها عدد فرضی ۱۵/۲۵ به عنوان معدل محاسبه شده و به صورت پیغام ارسال شده است. لازم به ذکر است که در این شکل برای سادگی نمایش فرض شده که تکرارهای حلقه برای سابقه‌های مختلف انجام شده است و طبیعتا استاندارد یوام‌ال برای آن به طور کامل رعایت نشده است.

در این بخش از طراحی لازم است به دو پرسش مهم پاسخ دهیم:

پرسش اول این است که در هر کدام از قسمت‌های طراحی که یک اکتور پیغام را فرستاده و منتظر جواب می‌ماند، آیا اکتور می‌تواند در طول مدت انتظار به فعالیت‌های دیگربپردازد؟ به عبارت بهتر، **آیا ارسال پیغام‌ها به صورت همگام است یا ناهمگام؟**

پرسش دوم این است که در صورتی که ارسال پیغام ناهمگام باشد ادامه‌ی فعالیت اکتور به چه صورتی مجاز است؟ آیا

می‌تواند پیغام‌های جدیدی دریافت کند و به اجرای منطق مربوط به آنها بپردازد؟

برای پاسخ به این پرسش‌ها در رویکرد اول، در هر مورد که پیغامی دریافت و فرستاده می‌شود این پرسش‌ها را بررسی می‌کنیم:

۱. اکتور دانشجو:

تنها پیغامی که اکتور دانشجو تا این مرحله از طراحی ارسال می‌کند پیغام GPAInfoRequest است. ابتدا منطق پیاده‌سازی شده در این تبادل این پیغام را بررسی می‌کنیم:

شبه کد ۸.۴ تبادل پیغام‌های دانشجو با اکتورهای سابقه را نشان می‌دهد. در این قطعه‌کد از دستور !? (تبادل همگام) برای فرستادن پیغام استفاده شده است. اکتور دانشجو به هر اکتور سابقه یک پیغام GPAInfoRequest می‌فرستد و با دریافت هر پاسخ GPAInfoResponse این عملیات را انجام می‌دهد: در صورتی که فیلد isForTerm از پیغام مقدار true داشته باشد مجموع وزن‌دار[۲۰] نمرات گرفته شده تا حال را با حاصل ضرب فیلد grade در فیلد units جمع می‌کند. و حاصل جمع واحدها را به اندازه‌ی units افزایش می‌دهد. نهایتا بعد از مبادله‌ی پیغام با تمام اکتورهای سابقه، حاصل تقسیم مجموع وزن‌دار نمرات بر تعداد واحدها به عنوان معدل دانشجو در ترم اعلام می‌شود.

حال پرسش اول برای اکتور دانشجو به این صورت بیان می‌شود:

آیا اکتور دانشجو بعد از ارسال پیغام GPAInfoRequest به یک اکتور سابقه و در مدتی که هنوز پاسخی از این اکتور دریافت نکرده می‌تواند به فعالیت خود ادامه دهد؟ ابتدا باید به این نکته دقت کرد که تفاوت اصلی رویکرد حاصل از پاسخ مثبت به این پرسش (ارسال ناهمگام) و پاسخ منفی به آن (ارسال همگام) از دیدگاه اکتور فرستنده‌ی درخواست چیست؟ با کمی دقت و تحلیل می‌توان دریافت که تفاوت اصلی این دو رویکرد از دیدگاه فرستنده در نحوه‌ی برخورد با پاسخ پیغام است. به بیان دقیق‌تر در حالت همگام، این که پاسخ دریافت شده مربوط به کدام درخواست بوده است، به طور ضمنی مشخص است. ولی اگر بعد از ارسال پیغام، اکتور منتظر جواب نماند و به کار خود ادامه دهد در هر زمان دیگری ممکن است پاسخ دریافت شود و در این هنگام امکان اینکه تشخیص داده شود این پاسخ مربوط به کدام درخواست بوده ممکن است امکان‌پذیر نباشد. دقت به منطق پیاده شده برای دریافت پیغام GPAInfoResponse نشان می‌دهد که اینکه هر پاسخ مربوط به کدام درخواست بوده اهمیتی ندارد. به بیان دیگر ترتیب دریافت این پاسخ‌ها تاثیری در معدل اعلام شده ندارد. بنابراین پاسخ به پرسش اول در مورد اکتور دانشجو مثبت است.

[۲۰] عددی که از جمع حاصل‌ضرب هر نمره در تعداد واحدهای درس حاصل شده است.

```
1  class Student(
2    var id: String,
3    var name: String,
4    var studyRecords: List[StudyRecord]) extends Actor {
5
6    var weightedSumOfGrades = 0
7    var sumOfUnits=0
8
9    override def act() {
10     loop {
11       react {
12        case GPARequest(term: Term) =>{
13          for(sr <- studyRecords) {
14            GPAInfoResponse(isForTerm:Boolean, grade: Double, units:Int) = sr !?
                  GPAInfoRequest(term)
15              if(isForTerm) {
16                weightedSumOfGrades += units * grade
17                sumOfUnits += sumOfUnits
18              }
19            }
20          }
21          sender ! GPAResponse(weightedSumOfGrades / sumOfUnits)
22        }
23        case TakeCourseRequest(offering:Offering)=>
24        ...
25       }
26     }
27   }
28 }
```

شکل ۸.۴: شبه‌کد اسکالا برای اکتور دانشجو در رویکرد ۱ با ارسال همگام پیغام

نتیجه: می‌توانیم پیغام‌های GPAInfoRequest را به صورت ناهمگام ارسال کنیم.

اکنون نوبت به پرسش دوم می‌رسد: آیا اکتور دانشجو در حالی که هنوز پاسخ تمام پیغام‌ها را دریافت نکرده می‌تواند درخواست جدیدی را پردازش کند؟

برای پاسخ به این پرسش فرض می‌کنیم که اکتور دانشجو در حالی که پاسخ تعدادی از پیغام‌های GPAInfoRequest را دریافت نکرده، یک پیغام جدید GPARequest دریافت می‌کند (یک درخواست جدید برای محاسبه‌ی معدل). برای محاسبه‌ی معدل، اکتور دانشجو مطابق منطق پیاده شده اقدام به ارسال پیغام GPAInfoRequest به تمام اکتورهای سابقه می‌کند. در این حالت فرض کنیم یک پیغام پاسخ GPAInfoResponse دریافت شود. با دریافت این پیغام باید متغیرهای محلی اکتور دانشجو به هدف محاسبه‌ی معدل بروزرسانی می‌شوند. اما با توجه به اینکه مشخص نیست که پاسخ دریافت شده مربوط به کدام درخواست بوده است نمی‌توانیم معدل را به صورت صحیح محاسبه کنیم. به عبارت دیگر منطق محاسبه‌ی معدل برای دو درخواست باهم مخلوط می‌شوند. به همین دلیل پاسخ به پرسش دوم منفی است.

نتیجه: علیرغم اینکه ارسال پیغام‌های GPAInfoRequest را می‌توانیم به صورت ناهمگام انجام دهیم (چون ترتیب دریافت پیغام‌ها اهمیتی ندارد)، قبل از دریافت همه‌ی پاسخ‌های مربوط به درخواست معدل درحال پردازش، نمی‌توانیم درخواست جدیدی دریافت کنیم.

البته باید دقت کرد که با وجود اینکه میزان به تعویق انداختن دریافت پاسخ‌ها محدود است (به دلیل پرسش دوم)، کماکان ارسال ناهمگام پیغام‌های GPAInfoRequest ارزشمند است. چرا که در حالت تبادل ناهمگام، تمام اکتورهای سابقه، به صورت همروند پاسخ این پیغام را آماده می‌کنند در حالی که در حالت همگام به صورت نوبتی و ترتیبی این اتفاق می‌افتد.

با توجه به پاسخ به این دو پرسش، طراحی اکتور دانشجو برای محاسبه‌ی معدل به صورت شبه‌کد شکل ۹.۴ تغییر می‌کند. در این شبه‌کد از روش تبادل پیغام آینده[۲۱] (رجوع کنید به بخش ۲.۲.۲) استفاده شده است.

[۲۱] Future

```
1  class Student(
2    var id: String,
3    var name: String,
4    var studyRecords: List[StudyRecord]) extends Actor {
5
6    var weightedSumOfGrades = 0
7    var sumOfUnits=0
8
9    override def act() {
10     loop {
11       react {
12        case GPARequest(term: Term) =>{
13          val replies = for(sr <- studyRecords) yield {sr !! GPAInfoRequest(term)}
14          for(i <- 0 until offerings.size) {
15            GPAInfoResponse(isForTerm:Boolean, grade: Double, units:Int) = replies(i)
16            if(isForTerm) {
17              weightedSumOfGrades += units * grade
18              sumOfUnits += sumOfUnits
19            }
20          }
21          sender ! GPAResponse(weightedSumOfGrades / sumOfUnits)
22        }
23        case TakeCourseRequest(offering:Offering)=>
24        ...
25       }
26     }
27   }
28 }
```

شکل ۹.۴: شبه‌کد اسکالا برای اکتور دانشجو در رویکرد ۱ با ارسال ناهمگام پیغام (آینده)

۲. اکتور سابقه:

در مورد اکتور سابقه جواب دادن به ۲ پرسش مذکور آسان‌تر است. این اکتور فقط پیغام GPAInfoRequest را ارسال می‌کند و با دریافت هر پیغام پاسخ GPAInfoResponse، صرفا نمره‌ی سابقه را به آن اضافه کرده و برای اکتور دانشجو ارسال می‌کند. واضح است که در این تبادل پیغام، ترتیب پیغام‌های پاسخ اهمیتی ندارد. بنابراین پاسخ اولین پرسش مثبت است (ارسال ناهمگام مجاز است). در مورد پرسش دوم با اینکه این اکتور هیچ حالتی[۲۲] برای درخواست‌ها نگه نمی‌دارد.[۲۳] اما دریافت درخواست جدید قبل از گرفتن پاسخ‌های درخواست قبلی مشکل دیگری ایجاد می‌کند. با توجه به اینکه هر درخواست که از اکتور دانشجو به اکتور سابقه می‌رسد، نهایتا باید توسط خود اکتور سابقه پاسخ داده شود، در هنگام فرستادن پیغام پاسخ باید آدرس فرستنده‌ی درخواست اولیه موجود باشد. در حالی که اگر قبل از پاسخ به درخواست اکتور دانشجو، درخواست جدیدی دریافت شود و عملیات پردازش درخواست جدید آغاز گردد، هیچ اثری از فرستنده‌ی درخواست اول برای ارسال پاسخ به آن موجود نخواهد بود. برای روشن شدن مطلب، شبه‌کد ۱۰.۴ را در نظر بگیرید که در آن فرض شده اکتور سابقه بتواند قبل از فرستادن پاسخ درخواست قبلی، درخواست جدیدی را پردازش کند. همان‌طور که در خط ۱۱ کد اشاره شده است، در هنگامی‌که یک پاسخ از اکتور ارائه دریافت شده، دسترسی به اکتور فرستنده‌ی پیغام اصلی (که در خط ۸ دریافت شده) وجود ندارد تا بتوانیم پاسخ را برای آن ارسال کنیم. باید دقت شود که با اینکه فرستنده‌ی یک پیغام به وسیله‌ی شیء sender قابل دسترسی است، اما این شیء به فرستنده‌ی پیغامی اشاره می‌کند که پیغام آن در حال پردازش است. در مورد خط ۱۱ این شیء اشاره به اکتور ارائه دارد که فرستنده‌ی آخرین پیغام بوده، نه اکتور دانشجو که در انتظار گرفتن پاسخ از اکتور سابقه است. بنابراین پاسخ به پرسش دوم در مورد اکتور سابقه منفی است و این اکتور باید پاسخ هر درخواست را قبل از پردازش درخواست‌های دیگر ارسال کند. نکته‌ی قابل توجه این است که با توجه به اینکه اکتور سابقه برای پاسخ به درخواست GPAInfoRequest تنها یک پیغام ارسال می‌کند و بدون دریافت پاسخ آن قادر به پاسخگویی به درخواست مذکور نیست، تفاوتی در ارسال همگام و ناهمگام پیغام وجود ندارد چرا که پس از ارسال تنها یک پیغام مجبور به توقف و انتظار برای دریافت پاسخ است. شبه کد ۱۱.۴ طراحی صحیح تبادل پیغام در اکتور سابقه را برای رویکرد ۱ نشان می‌دهد.

---

[۲۲] state

[۲۳] بر خلاف حالت اکتور دانشجو که در آن متغیرهایی برای هر درخواست مقداردهی می‌شدند.

```
1
2 class StudyRecord(
3   var grade: Double,
4   var offering: Offering) extends Actor {
5   override def act() {
6     loop {
7       react {
8        case GPAInfoRequest(term: Term) => //comes from student
9          offering ! GPAInfoRequest(term)
10       case GPAInfoResponse(isForTerm, grade, units) => //comes from offering
11         who ! GPAInfoResponse(...) ????
12      }
13    }
14  }
15 }
```

شکل ۱۰.۴: شبه‌کد اکتور سابقه برای حالتی که بتواند قبل از پاسخ به درخواست قبلی، درخواست جدیدی را پردازش کند. (این رویکرد اشتباه است.)

```
1
2 class StudyRecord(
3   var grade: Double,
4   var offering: Offering) extends Actor {
5   override def act() {
6     loop {
7       react {
8        case GPAInfoRequest(term: Term) => //comes from student
9           val firstSender = sender
10          offering !? GPAInfoRequest(term) match {
11            case GPAInfoResponse(isForTerm,null,units)
12            firstSender ! GPAInfoResponse
13          }
14      }
15    }
16  }
17 }
```

شکل ۱۱.۴: شبه‌کد صحیح برای اکتور سابقه در رویکرد ۱

۳. اکتور ارائه:

اکتور ارائه پس از دریافت درخواست GPAInfoRequest دو پیغام به ترتیب برای اکتورهای ترم و درس ارسال می‌کند و در هر کدام از این دو پیغام بخشی از اطلاعات لازم برای فرستادن پاسخ به اکتور سابقه را از آنها دریافت می‌کند. پرسش اول در مورد اکتور ارائه اینطور مطرح می‌شود که آیا اکتور ارائه پس از فرستادن هر کدام از پیغام‌های مذکور به ترم و درس می‌تواند پیغام بعدی را ارسال کند یا باید پس از ارسال هرکدام بلافاصله منتظر دریافت پاسخ بماند؟ جواب این پرسش مثبت است به این دلیل که ترتیب پیغام‌های پاسخ اهمیتی ندارد. اما با استدلالی مشابه آنچه که در مورد اکتور سابقه توضیح داده شد، جواب پرسش دوم برای اکتور ارائه منفی است. یعنی اکتور ارائه تا زمانی که پاسخ یک درخواست را به اکتور سابقه‌ی مربوطه نفرستاده، نمی‌تواند درخواست جدیدی (احتمالاً از یک اکتور سابقه‌ی دیگر) پردازش کند. به همین دلیل حداکثر میزان ناهمگامی در ارسال پیغام‌ها برای اکتور ارائه این است که دو پیغام IsYourTermRequest و NumOfUnitsRequest را به صورت ناهمگام برای اکتورهای ترم و درس ارسال کند و سپس منتظر دریافت پاسخ آنها بماند. بنابراین طراحی تبادل پیغام اکتور ارائه در رویکرد ۱ مطابق شبه‌کد شکل ۱۲.۴ خواهد بود. در این شکل نیز از ویژگی آینده[۲۴] (رجوع کنید به ۲.۲.۲) استفاده شده است.

[۲۴] Future

```
1 class Offering(
2   var id: String,
3   var course: Course,
4   var examDate: Date,
5   var term: Term) extends Actor {
6   override def act() {
7     loop {
8       react {
9        case GPAInfoRequest(gpaTerm: Term) =>
10         val termFuture = term !! IsYourTermRequest(gpaTerm)
11         val courseFuture = course !! NumOfUnitsRequest
12         sender ! GPAInfoResponse(termFuture(),null,courseFuture())
13         }
14       }
15     }
16   }
17 }
```

شکل ۱۲.۴: شبه‌کد طراحی نحوه‌ی تبادل پیغام برای اکتور ارائه در رویکرد ۱.

۴. اکتورهای ترم و درس:

در مورد این دو اکتور تصمیم به استفاده از ارسال همگام یا ناهمگام بسیار ساده است. با توجه به اینکه در هر دو اکتور مذکور، تمام اطلاعات لازم برای پاسخ به درخواست‌ها در خود اکتور موجود است، نیازی به ارسال پیغام به سایر اکتورها وجود ندارد و پاسخ درخواست‌ها بلافاصله ارسال می‌شود. لذا هیچ نیازی به تبادل همگام وجود ندارد (چون پاسخی دریافت نخواهد شد). طراحی این دو اکتور از نظر تبادل پیغام در شبه‌کدهای ۱۳.۴ و ۱۴.۴ نمایش داده شده است.

```
1 class Term(
2   var name: String,
3   var startDate: Date,
4   var offerings: List[Offering]) extends Actor {
5   override def act() {
6     loop {
7       react {
8        case IsYourTermRequest(gpaTerm) =>
9          sender ! (gpaTerm.name == name)
10      }
11    }
12  }
13 }
```

شکل ۱۳.۴: شبه‌کد طراحی نحوه‌ی تبادل پیغام برای اکتور ترم در رویکرد ۱.

```
1 class Course(
2   var id: String,
3   var name: String,
4   var units: Int,
5   var preRequisites: List[Course]) extends Actor {
6   override def act() {
7     loop {
8       react {
9        case NumOfUnitsRequest =>
10         sender ! units
11      }
12    }
13  }
14 }
```

شکل ۱۴.۴: شبه‌کد طراحی نحوه‌ی تبادل پیغام برای اکتور درس در رویکرد ۱.

۲.۲.۳.۴ رویکرد دوم

رویکرد دوم از طراحی مورد کاربرد محاسبه‌ی مدل را با بررسی رویکرد ۱ و طرح چند پرسش در مورد آن آغاز می‌کنیم. نحوه‌ی طراحی ارتباطات و پیغام‌ها در رویکرد اول در بخش قبل به طور کامل توضیح داده شد. در این قسمت خلاصه‌ای از این طراحی را بررسی می‌کنیم:

عملیات با دریافت پیغام درخواست معدل GPARequest(term) در اکتور دانشجو آغاز می‌شود. اکتور دانشجو به هر کدام از اکتورهای سابقه، یک پیغام درخواست اطلاعات معدل GPAInfoRequest(term) ارسال می‌کند. این پیغام از طریق اکتور سابقه به دست اکتور ارائه می‌رسد و از طریق این اکتور به دست اکتورهای درس و ترم می‌رسد و هر کدام از این اکتورها اطلاعات لازم را برای اکتور ارائه ارسال می‌کنند. در ادامه اکتور ارائه یک پیغام پاسخ اطلاعات معدل (GPAInfoResponse) تولید می‌کند و برای اکتور سابقه ارسال می‌کند. سابقه عدد نمره را به پیغام اضافه کرده و برای دانشجو می‌فرستد. دانشجو با تکرار همین عملیات برای تمام سابقه‌ها تمام اطلاعات لازم برای محاسبه‌ی معدل در اختیار دارد.

هر اکتور در این مورد کاربرد به دلایل مختلفی اقدام به مشارکت در محاسبه‌ی معدل می‌کند: دانشجو به این دلیل که مسئولیتِ گرفتن درخواست اصلی را دارد و نیز به این دلیل که به اکتور سابقه دسترسی دارد. اکتور سابقه به این دلیل که نمره (یکی از اطلاعات لازم برای محاسبه‌ی معدل) را در اختیار دارد و نیز از طریق اکتور ارائه به درس و ترم دسترسی دارد. اکتور ارائه به دلیل دسترسی به درس و ترم. و اکتورهای درس و ترم به دلیل اینکه اطلاعات مورد نیاز برای محاسبه‌ی معدل را در اختیار دارند. در نتیجه مشارکت تمام این اکتورها در محاسبه‌ی معدل ضروری است. اما پرسشی که پیش می‌آید این است که آیا میزان مشارکت این اکتورها نیز باید در همین میزان باشد؟ اگر هر دریافت یا ارسال یک نوع پیغام را یک مشارکت برای اکتور در طراحی این مورد کاربرد در نظر بگیریم، آیا می‌توان تعداد مشارکت‌های اکتورها را کاهش داد؟ به عنوان مثال اکتور سابقه را در نظر می‌گیریم. همان‌طور که ذکر شد مشارکت این اکتور به دلیل داشتن فیلد نمره و نیز دسترسی به اکتور ارائه ضروری است. تعداد مشارکت اکتور سابقه با توجه به تعریف ارائه شده، از روی نمودار ترتیب شکل ۷.۴ به این ترتیب قابل استخراج است: هر فلشی که از خط زمان[۲۵] اکتور سابقه خارج یا به آن وارد می‌شود معادل ارسال یا دریافت یک نوع پیغام است. بنابراین تعداد مشارکت اکتور سابقه در این مورد کاربرد ۴ است. مشارکت اول مربوط به دریافت پیغامِ درخواست از دانشجو است، مشارکت دوم مربوط به ارسال درخواست به ارائه است، مشارکت سوم دریافت پاسخ از ارائه و مشارکت چهارم مربوط به ارسال پاسخ به دانشجو است. حال بررسی می‌کنیم که از این تعداد مشارکت، دو مورد الزامی است. یکی دریافت درخواست از دانشجو به دلیل اینکه دانشجو از طریق دیگری به اطلاعات

[۲۵] time line

مورد نیاز برای محاسبه‌ی معدل دسترسی ندارد، و دیگری ارسال درخواست برای ارائه. دو مورد دیگر یعنی دریافت پاسخ ارائه و تحویل آن به دانشجو را می‌توان حذف کرد. روش حذف به این صورت است که اکتور ارائه به نحوی مطلع شود که جواب نهایی به چه کسی ارسال خواهد شد (دانشجو). این کار از طریق **قرار دادن مقصد نهایی پیغام در داخل پیغام** قابل انجام است. در این حالت دیگر نیازی به برگشت پیغام به دست سابقه وجود ندارد. تنها موردی که به نظر مشکل‌ساز می‌آید این است که فیلد نمره در رویکرد ۱ در هنگام برگشت پیغام در آن قرار داده می‌شود و اگر پیغام از طریق سابقه برگشت داده نشود فیلد نمره را نخواهد داشت. البته این مورد به سادگی قابل حل است و در همان بار اول که پیغام به دست سابقه رسید، می‌تواند نمره را به پیغام اضافه کند. البته مثال اکتور سابقه در مورد بقیه‌ی اکتورها نیز قابل بررسی است ولی به دلیل پرهیز از تکرار استدلال به همین مورد اکتفا می‌کنیم.

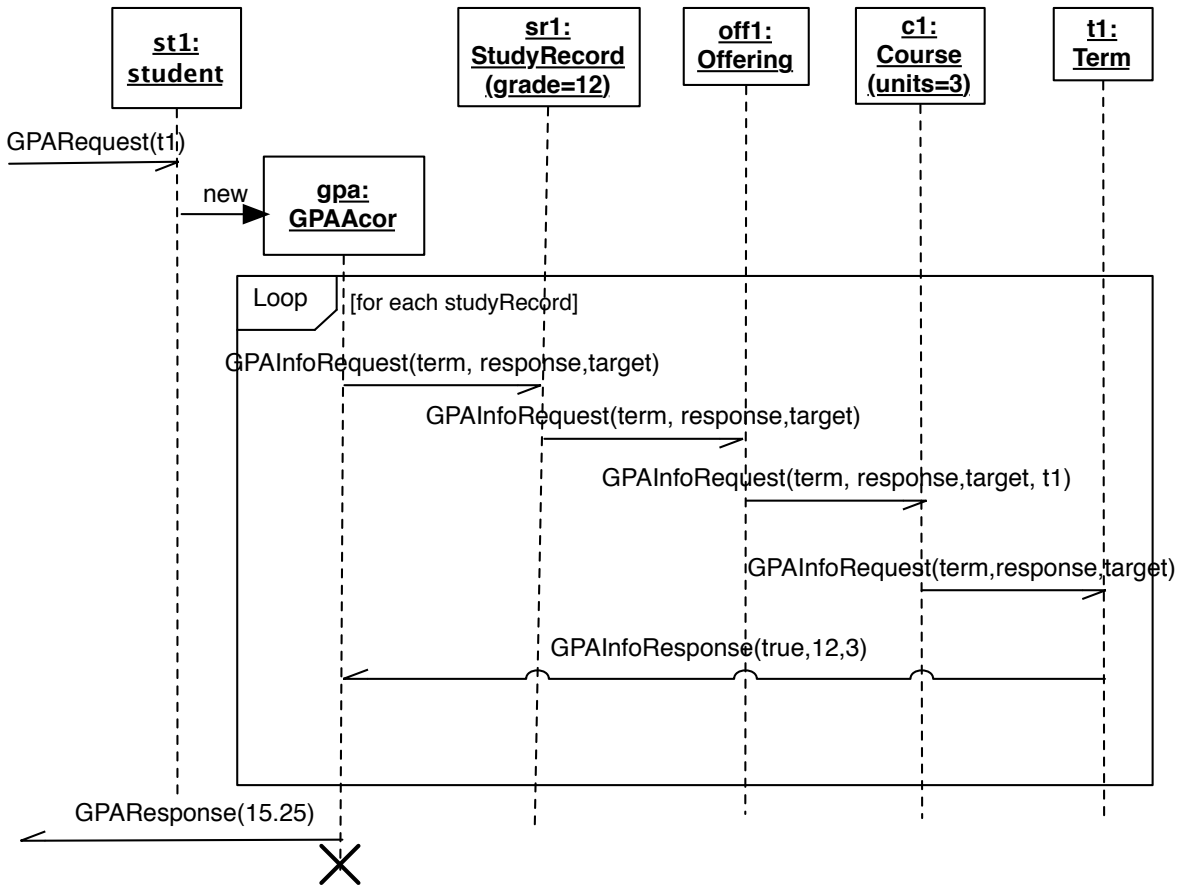
مورد دیگری که در رویکرد ۱ بررسی می‌کنیم عدم امکان پردازش درخواست‌های جدید در هنگام انتظار برای تکمیل اطلاعات مورد نیاز برای پاسخ به درخواست قبلی است. مثلا در مورد دانشجو این مورد باعث شد که در رویکرد ۱، دانشجو قبل از ارسال پاسخ درخواست معدل، درخواست دیگری را بررسی کند. در مورد دانشجو دلیل این پدیده این بود که منطق محاسبه‌ی معدل قسمتی از حالت[۲۶] این اکتور بود و تداخل درخواست‌های معدل می‌تواند باعث عملکرد غلط اکتور شود. یک راه برای حل این مشکل این است که به نوعی مشخص کنیم که هر پاسخی که اکتور دانشجو دریافت می‌کند مربوط به کدام درخواست اصلی بوده است. یعنی حالت اکتور را در قالب نگاشت‌هایی از پیغام‌ها حفظ کنیم. مثلا برای اکتور دانشجو، به جای اینکه یک متغیر برای مجموع نمره‌هایی که تا این لحظه پاسخ آنها بررسی شده (رجوع کنید به شبه‌کد شکل ۹.۴)، می‌توانیم نگاشتی[۲۷] از شناسه‌ی درخواست معدل به متغیر مجموع نگهداری کنیم، به این ترتیب با رسیدن یک پاسخ، متغیر مربوط به درخواست مربوطه برای محاسبه استفاده می‌شود. البته این روش اولاً باعث پیچیده‌تر شدن منطق اکتور می‌شود و ثانیاً نگهداری ساختار داده‌ی نگاشت اهمیت زیادی پیدا می‌کند. به این دلایل استفاده از نگاشت رویکرد مناسبی نیست. روش دیگر این است که به حالت مربوط به بررسی یک درخواست را به اکتور دیگری که به همین منظور ایجاد می‌شود، منتقل کنیم. مثلا وقتی دانشجو یک درخواست محاسبه‌ی معدل دریافت می‌کند، یک اکتور مختص همان درخواست ایجاد کنیم و همه‌ی تبادلات مربوط به آن درخواست را به اکتور جدید واگذار کنیم. طبیعتا تمام اطلاعات لازم از جمله دسترسی به اکتور سابقه باید به اکتور جدید منتقل شود. در نتیجه‌ی این رویکرد، دانشجو می‌تواند با دریافت هر درخواست معدل بلافاصله به پردازش آن بپردازد.

با توجه به موارد ذکر شده و بدون تکرار نکاتی که در رویکرد اول ذکر شد به ارائه‌ی خلاصه‌ای از طراحی اکتورها در

---

[۲۶] state

[۲۷] map



شکل ۱۵.۴: نمودار ترتیب برای رویکرد دوم محاسبه‌ی معدل

رویکرد دوم می‌پردازیم. شکل ۱۵.۴ نمودار ترتیب برای رویکرد دوم محاسبه‌ی معدل را نشان می‌دهد. برای پرهیز از تکرار، در این رویکرد مراحل طراحی معرفی شده در رویکرد اول بسط داده نشده است و صرفاً چند تغییر اساسی توضیح داده می‌شود.

۱. اکتور محاسبه‌ی معدل (GPAActor):

همان طور که قبلا توضیح داده شد، این اکتور برای انجام کل فعالیت‌های مربوط به یک درخواست معدل را انجام می‌دهد (در رویکرد اول این کار توسط خود اکتور دانشجو انجام می‌شد). این اکتور برای انجام وظیفه‌ی خود اولاً نیاز به برقراری ارتباط با اکتورهای سابقه دارد، و ثانیاً نیاز به دسترسی به مقصد پاسخ درخواست دارد تا بتواند نتیجه را برای آن ارسال کند. این موارد توسط اکتور دانشجو در اختیار اکتور محاسبه‌ی معدل قرار می‌گیرد. شبه کد ۱۶.۴ نحوه‌ی طراحی این اکتور را نشان می‌دهد. اکتور محاسبه‌ی معدل با شروع به کار پیغام‌های لازم برای

سایر اکتورها را ارسال می‌کند و با گرفتن هر پاسخ، متغیرهای حالت خود را بروزرسانی می‌کند. پایان کار این اکتور زمانی مشخص می‌شود که به تعدادی که پیغام ارسال کرده پاسخ دریافت کند. این تعداد برابر با تعداد اکتورهای سابقه است. بنابراین پس از دریافت این تعداد پیغام، معدل محاسبه شده را برای مقصد نهایی ارسال می‌کند.

تغییر مهم اکتور دانشجو این است که با توجه به واگذاری عملیات محاسبه‌ی معدل به اکتوری دیگر، نیازی به نگهداری متغیرهای حالت که به این منظور ایجاد شده بودند، ندارد. شبه کد اکتور دانشجو در رویکرد جدید در شکل ۱۷.۴ نشان داده شده است. مقایسه‌ی طراحی این اکتور در دو رویکرد نشان می‌دهد که با انجام این عمل، طراحی اکتور دانشجو بسیار ساده‌تر شده است.

```
1 class GPAActor(
2   val term: Term,
3   val studyRecords: List[StudyRecord],
4    val target: Actor) extends Actor {
5
6   var processedMessages = 0
7   var weightedSumOfGrades = 0
8   var sumOfUnits = 0
9
10  override def act() {
11    for(sr <- studyRecords)
12      sr ! GPAInfoRequest(term, this, )
13
14    loop {
15      react {
16        case GPAInfoResponse(isForTerm:Boolean, grade: Double, units:Int) =>
17          processMessage(isForTerm, grade, units)
18      }
19    }
20  }
21
22  def processMessage(isForTerm:Boolean, grade:Double, units:Int) {
23    if(isForTerm) {
24            weightedSumOfGrades += units * grade
25            sumOfUnits += sumOfUnits
26    }
27    processedMessages ++
28    if(processedMessages == studyRecords.size) {
29      target ! GPAResponse(weightedSumOfGrades / sumOfUnits)
30      exit
31    }
32  }
33 }
```

شکل ۱۶.۴: شبه‌کد طراحی اکتور محاسبه‌ی معدل در رویکرد ۲.

```
1 class Student(
2   var id: String,
3   var name: String,
4   var studyRecords: List[StudyRecord]) extends Actor {
5
6   override def act() {
7     loop {
8       react {
9        case GPARequest(term: Term) =>
10        val gpa = new GPAActor(term, studyRecords, sender)
11        gpa.start
12       case TakeCourseRequest(offering:Offering)=>
13       ...
14      }
15    }
16  }
17 }
```

شکل ۱۷.۴: شبه‌کد طراحی اکتور دانشجو در رویکرد ۲.

۳.۲.۳.۴ مقایسه‌ی دو رویکرد

در بخش‌های قبلی ۲ رویکرد متخلف برای طراحی اکتورها در ارتباط با مورد کاربرد محاسبه‌ی معدل معرفی شده و مراحل انجام طراحی در آنها شرح داده شد. علیرغم صحت عملکرد هر دو رویکرد، تفاوت‌های کیفی در طراحی به وسیله‌ی این دو رویکرد حائز اهمیت هستند. به همین دلیل در این بخش به مقایسه‌ی این دو رویکرد می‌پردازیم.
رویکرد دوم دو تغییر عمده نسبت به رویکرد اول دارد:

۱. قرار دادن مقصد نهایی درخواست در داخل پیغام:

در رویکرد اول هر اکتوری که پیغامی را به عنوان درخواست از یک اکتور دیگر دریافت می‌کند، وظیفه‌ی پاسخ به آن را نیز به عهده دارد. در صورتی که برای پاسخ به درخواست نیاز به برقراری ارتباط با اکتورهای دیگر وجود داشته باشد،این اکتور اقدام به ارسال پیغام‌های مرتبط به سایر اکتورها می‌کند و در نهایت با جمع‌آوری پاسخ‌ها، درخواست اصلی را پاسخ می‌دهد. با اینکه این رویکرد از دیدگاه طراحی شیءگرا به روش ترتیبی، رویکردی متداول و حتی اجباری است[۲۸]، در مدل تبادل پیغام این امکان وجود دارد که پاسخ درخواست را اکتوری غیر از دریافت کننده‌ی درخواست ارسال کند. لازم به ذکر است که در مدل اکتور هیچ فرضی در مورد مشخصات فرستنده‌ی پیغام صورت نمی‌گیرد. بنابراین یک اکتور می‌تواند به جای اینکه پس از ارسال پیغام‌های مربوط به یک درخواست، منتظر دریافت جواب برای فرستادن به درخواست کننده بماند، آدرس (نام) مقصد نهایی را در داخل پیغام برای اکتور ها ارسال کند تا در صورت لزوم از آن برای فرستادن نتیجه استفاده کنند. رویکرد دوم در واقع از این امتیاز استفاده کرده و به این روش از تعدادی از تبادلات پیغام که صرفاً به دلیل ذکر شده صورت می‌گیرند، جلوگیری می‌کند. با این کار نیازی به برگشت پیغام در همان مسیری که طی شده وجود نخواهد داشت و در هر لحظه که اطلاعات لازم برای تکمیل پاسخ تأمین شود، پاسخ به مقصد ارسال خواهد شد.

۲. واگذار کردن پردازش‌های مربوط به یک درخواست به یک اکتور موقت:

در رویکرد اول اکتور دانشجو، پس از ارسال پیغام‌های لازم و دریافت جواب، تمام محاسبات لازم برای تعیین معدل را انجام می‌داد. در اثر استفاده از این رویکرد، اولاً دانشجو باید تعدادی پیغام برای تهیه‌ی اطلاعات لازم جهت محاسبه‌ی معدل به سایر اکتورها ارسال کرده و منتظر جواب بماند، ثانیاً برای محاسبه‌ی معدل اطلاعات موقتی را به عنوان متغیر حالت در خود نگهداری کند. مقدار این متغیرها فقط در زمانی که یک درخواست مشخص

[۲۸] در طراحی شیءگرای ترتیبی، مکانیزم کنترل برنامه فراخوانی متد است. با هر فراخوانی متد، منطق پیاده شده در متد اجرا می‌شود و پس از بازگشت از متد، اجبارا کنترل برنامه به همان قسمتی که متد فراخوانی شده بود برمی‌گردد.

در حال پردازش است معتبر است به همین دلیل در صورت شروع به پردازش درخواست‌های دیگر قبل از اتمام عملیات مربوط به درخواست قبلی امکان‌پذیر نمی‌باشد. در نتیجه میزان همروندی در درخواست‌های مشابه پایین می‌آید. از طرف دیگر در صورتی که قرار باشد، اکتور انواع متعددی از درخواست‌هایی را که این خاصیت را دارند پردازش کند، مدیریت پیچیدگی حاصل از اطلاعات حالت مربوط به درخواست‌های مختلف نیز کار آسانی نخواهد بود و منجر به پیچیدگی زیاد و تغییرپذیری کمتر کلاس خواهد شد. به همین دلایل در رویکرد دوم سیاست جدید اتخاذ شد و آن سپردن کل فعالیت‌های محاسبه‌ی معدل به یک اکتور جدید است. با این کار دو نتیجه‌ی مطلوب حاصل می‌شود. اولاً پیچیدگی‌های مربوط به اجرای یک درخواست به اکتور دیگری منتقل می‌شود که صرفاً برای پاسخ به درخواست مورد نظر طراحی شده است. ثانیا با توجه به اینکه هر نمونه از اکتور جدید صرفاً محدود به یک درخواست بوده و پس از پاسخ به آن به فعالیت خاتمه می‌دهد، امکان پاسخ به درخواست‌های همروند به درخواست‌ها هم به وجود می‌آید.

لازم به ذکر است که هدف از معرفی این دو رویکرد در طراحی منطق مربوط به محاسبه‌ی معدل صرفاً تأکید بر تفاوت‌های آنها و حفظ وضوح روش طراحی دارد. طبیعتاً علیرغم صحت رویکرد اول، در ادامه‌ی طراحی از سیاست‌های ذکر شده در رویکرد دوم استفاده خواهد شد

۳.۳.۴ مورد کاربرد اخذ درس

در بخش قبل مراحل طراحی مورد کاربرد محاسبه‌ی معدل با استفاده از دو رویکرد مختلف توضیح داده شد. در این بخش مراحل طراحی مورد کاربرد اخذ درس با توجه به تجربیات حاصل از بخش قبل ارائه می‌گردد.

توصیف مورد کاربرد اخذ درس در جدول ۲.۴ ارائه شد. دانشجو در زمان انتخاب واحد یکی از ارائه[۲۹]های موجود ترم را انتخاب می‌کند. سیستم شرایط لازم برای اخذ این ارائه را بررسی می‌کند. در صورتی که دانشجو مجاز به انتخاب این ارائه باشد، یک سابقه از ارائه‌ی مورد نظر را برای دانشجو ذخیره می‌کند. در صورتی که هر کدام از شرایط لازم برای اخذ محقق نشده باشد سیستم یک پیغام خطا برای کاربر نمایش می‌دهد. همانند مورد کاربرد قبل، این مورد کاربرد هم با دریافت یک پیغام توسط اکتور دانشجو آغاز می‌شود. تنها اطلاعاتی که در این پیغام باید موجود باشد ارائه‌ی انتخاب شده برای اخذ است. بنابراین فرمت پیغام درخواست اخذ درس به شکل زیر خواهد بود:

TakeCourseRequest(offering: Offering)

[۲۹] Offering

پاسخ این درخواست نیز باید حاوی نتیجه‌ی عملیات و نیز احتمالاً یک پیغام برای کاربر خواهد بود. بنابراین پیغام پاسخ اخذ درس به فرمت زیر خواهد بود:

TakeCourseResponse(result: Boolean, comment: String)

مطابق توضیحاتی که در طراحی مورد کاربرد محاسبه‌ی معدل داده شد، اکتور دانشجو در مواجهه با پیغام درخواست اخذ دو راهکار کلی پیش رو دارد. راهکار اول این است که منطق مورد نیاز برای پردازش اخذ درس را خودش پیاده‌سازی کند (مانند رویکرد اول در طراحی مورد کاربرد محاسبه‌ی معدل) و راهکار دوم این است که به یک اکتور دیگر وکالت این محاسبات را بسپارد. همان‌طور که در بخش قبل ذکر شد، تصمیم به سپردن محاسبات به کاربرد دیگر به دو انگیزه‌ی مختلف صورت می‌گیرد. انگیزه‌ی اول جلوگیری از پیچیده و بزرگ شدن یک اکتور در اثر پردازش پیغام‌های مختلف و انگیزه‌ی دوم ایجاد امکان همروندی در پردازش پیغام‌های مشابه.

در این مورد کاربرد هر دو انگیزه برای سپردن محاسبات به یک اکتور دیگر معتبر می‌باشند: اکتور دانشجو در مدل دامنه‌ی معرفی شده، مسئولیتِ دریافت اکثر درخواست‌های کاربران را به عهده دارد (به دلیل اینکه کاربر اصلی این سیستم دانشجو است)، درخواست‌های مختلفی را دریافت خواهد کرد. به همین دلیل در صورتی که پردازش تمام این پیغام‌ها را بر عهده بگیرد، اندازه و پیچیدگی آن زیاد شده و در نتیجه تغییرپذیری آن تنزّل خواهد کرد (انگیزه‌ی اول). علاوه بر این، اکتور دانشجو برای پردازش هر درخواست اخذ درس، باید شروط مختلفی را بررسی کند و برای این کار با اکتورهای دیگر به دفعات تبادل پیغام انجام خواهد داد و برای حفظ نتایج میانی تبادلات پیغام تا پایان پردازش درخواست، مجبور به استفاده از متغیرهای حالت اکتور (فیلدهای داده‌ی محلی) خواهد بود (مشابه متغیرهایی که در محاسبه‌ی معدل استفاده شد). در نتیجه پردازش همروند درخواست‌های اخذ درس بسیار پیچیده و یا نشدنی خواهد بود. بنابراین در این مورد، ایجاد همروندی در پردازش درخواست‌ها نیز انگیزه‌ی معتبری برای سپردن محاسبات به یک اکتور دیگر است (انگیزه‌ی دوم).

۱.۳.۳.۴ اکتور اخذ درس

با توجه به توضیحات ذکر شده اکتور دانشجو با گرفتن درخواست اخذ درس، کلیه‌ی محاسبات لازم و ارسال پاسخ را به اکتور اخذ درس منتقل می‌کند. وظیفه‌ی اکتور اخذ درس بررسی شرایط دانشجو برای اخذ درس و ارسال پاسخ درخواست است. طبق توصیف مورد کاربرد اخذ درس (جدول ۲.۴)، شروطی که باید قبل از قبول اخذ درس بررسی

شوند عبارتند از:

۱. دانشجو در ترم‌های قبل درس مربوط به ارائه‌ی انتخاب شده را نگذرانده باشد.

۲. دانشجو در ترم جاری این درس را اخذ نکرده باشد.

۳. دانشجو تمام پیش‌نیازهای این درس را با موفقیت گذرانده باشد.

۴. تعداد واحدهای اخذ شده توسط دانشجو در این ترم پس از اخذ این درس بیشتر از ۲۰ نشود.

در این مرحله، اکتور اخذ درس باید برای هریک از شروط ذکر شده، اولاً قالب پیغام مناسب را طراحی کند، ثانیاً مقصد پیغام را مشخص کند (تشخیص اکتور مسئول). این دو مورد باید برای هریک از چهار شرط فوق بررسی شوند. در ادامه بررسی این موارد برای شرط اول به صورت مبسوط بررسی می‌شود و برای سایر شروط با توجه به شباهت به شرط اول صرفاً نتیجه‌ی بررسی ارائه می‌گردد:

- شرط ۱:

این شرط باید تعیین کند که دانشجو قبلاً سابقه‌ای از گذراندن این درس را دارد یا خیر. قبل از انتخاب قالب پیغام، بحثی در مورد پذیرنده‌ی پیغام (اکتور مسئول) می‌کنیم. گزینه‌های موجود برای اکتور مسئول بررسی شرط گذراندن درس باشد اینها هستند:

۱. خود اکتور اخذ درس:

انتخاب اول در واقع به این معنی است که اکتور اخذ درس به جای اینکه درخواستی برای بررسی گذرانده شدن درس ارسال کند، خود این بررسی را به عهده بگیرد. البته این به این معنی نیست که برای انجام این بررسی هیچ پیغامی به اکتورهای دیگر ارسال نکند، بلکه به این معنی است که وظیفه‌ی پیاده‌سازی منطق لازم برای رسیدن به پاسخ این پرسش (آیا این دانشجو قبلاً این درس را گذرانده است؟) بر عهده‌ی اکتور اخذ درس باشد. این حالت به دو دلیل مناسب نیست: اولاً در این حالت اکتور اخذ درس به صورت ابتدا به ساکن (بدون دریافت درخواستی برای این کار) اقدام به پیاده‌سازی یک منطق کرده است. در نتیجه این پیاده‌سازی به جز این اکتور برای اکتور دیگری قابل استفاده‌ی مجدد نیست. [۳۰] ثانیاً با توجه به اینکه

[۳۰] البته یک راهکار ممکن برای برطرف کردن این مشکل این است که به اکتور اخذ درس، قابلیت دریافت پیغامی از نوع بررسی شرط مذکور اضافه شود و این اکتور برای بررسی این شرط، یک پیغام به خودش بفرستد تا به این صورت قابلیت استفاده‌ی مجدد داشته باشد. اما ایراد این رویکرد این است که از نظر منطقی اضافه کردن این رفتار به اکتوری که صرفاً وظیفه‌ی پاسخ به یک درخواست اخذ درس را دارد، از نظر تقسیم مسئولیت عمل درستی نیست.

این اکتور شروط متعددی را بررسی می‌کند، پیاده کردن منطق بررسی این شروط خوانایی کلاس را کاهش می‌دهد.

البته یک راهکار ممکن برای برطرف کردن این مشکل این است که به اکتور اخذ درس، قابلیت دریافت پیغامی از نوع بررسی شرط مذکور اضافه شود و این اکتور برای بررسی این شرط، یک پیغام به خودش بفرستد تا به این صورت قابلیت استفاده‌ی مجدد داشته باشد. اما ایراد این رویکرد این است که از نظر منطقی اضافه کردن این رفتار به اکتوری که صرفاً وظیفه‌ی پاسخ به یک درخواست اخذ درس را دارد، از نظر تقسیم مسئولیت عمل درستی نیست.

۲. اکتور جدیدی که به این منظور تولید می‌شود:

این رویکرد ایرادهای شمرده شده برای انتخاب اول را ندارد. اما با فرض این که درخواست بررسی گذرانده شدن درس یک درخواست قابل استفاده‌ی مجدد در منطق دامنه‌ی سیستم باشد، با این رویکرد در هر قسمتی از برنامه که نیاز به بررسی این درخواست وجود داشته باشد، باید اکتوری به این منظور ایجاد شود و اطلاعات لازم به آن داده شود و سپس درخواست برای آن ارسال شود. از نظر طراحی شیءگرا، تکرار این عملیات در هر بار نیاز به این درخواست پدیده‌ی مطلوبی نمی‌باشد.

۳. اکتور دانشجو:

انتخاب اکتور دانشجو برای ارسال درخواست بررسی گذرانده شدن درس علاوه بر اینکه ایرادهای مطرح شده در گزینه‌ی اول را ندارد، مشکل تکرار عملیات (گزینه‌ی دوم) را نیز ندارد. در این حالت، هر اکتوری که نیاز به بررسی درخواست گذرانده شدن درس را داشته باشد، پیغام مربوطه را برای اکتور دانشجو ارسال می‌کند و تنها جایی که عملیات ایجاد اکتور جدید برای پردازش این درخواست انجام می‌شود اکتور دانشجو است. از نظر منطق دامنه نیز بررسی گذرانده شدن درس توسط اکتور دانشجو انتخاب مطلوبی به نظر می‌رسد.

با توجه به استدلال فوق، اکتور اخذ درس، اکتور دانشجو را به عنوان مسئول بررسی گذرانده شدن درس انتخاب می‌کند.

با انتخاب مقصد پیغام درخواست بررسی گذرانده شدن درس، طراحی قالب پیغام آن به آسانی انجام می‌شود. با توجه به اینکه این پیغام به مقصد اکتور دانشجو ارسال می‌شود، تنها داده‌ای که لازم است در آن قرار داده شود درس مربوطه است. بنابراین قالب پیغام درخواست به صورت زیر می‌باشد:

PassedRequest(course:Course)

پیغام پاسخ کافی است که اطلاع دهد که درس مورد نظر گذرانده شده است یا خیر. بنابراین قالب پیغام پاسخ به

صورت زیر می‌باشد:

PassedResponse(result:Boolean)

- شرط ۲:

این شرط باید تعیین کند که دانشجو قبلاً در همین ترم این درس را اخذ کرده است یا خیر. با استدلال مشابه شرط ۱ به این نتیجه می‌رسیم که مقصد پیغام درخواست بررسی اخذ تکراری اکتور دانشجو است و قالب پیغام‌های درخواست و پاسخ برای این شرط به صورت زیر می‌باشد:

TakenRequest(course:Course) TakenResponse(result:Boolean)

- شرط ۳:

این شرط باید تعیین کند که دانشجو تمام پیش‌نیازهای درس را با موفقیت گذرانده است یا خیر. با استدلال مشابه شرط ۱ به این نتیجه می‌رسیم که مقصد پیغام درخواست بررسی اخذ تکراری اکتور دانشجو است و قالب پیغام‌های درخواست و پاسخ برای این شرط به صورت زیر می‌باشد:

PassedPresRequest(course:Course) PassedPresResponse(result:Boolean)

- شرط ۴:

این شرط کنترل می‌کند که با اخذ این درس آیا تعداد واحدهای دانشجو در ترم جاری بیشتر از ۲۰ می‌شود یا خیر. گذراندن بیش از تعدادی واحد به احتمال زیاد در چنین سیستمی به جز در بررسی شرایط کاربرد دیگری ندارد (بر خلاف موردی مثل بررسی گذرانده شدن درس که در موارد متعددی می‌تواند کاربرد داشته باشد) به همین دلیل پیغام مربوط به این مورد بهتر است به جای دانشجو به اکتور دیگری که مختص این کاربرد طراحی می‌شود سپرده شود. گیرنده‌ی پیغام مربوط به این شرط اکتور تأیید تعداد واحد (UnitsValidatorActor) خواهد بود. با توجه به این که اکتور مذکور نیاز به دسترسی به سوابق دانشجو و نیز تعداد واحد درس انتخاب شده دارد، در هنگام ایجاد این اکتور، باید فیلدهای دانشجو و درس را در اختیار اکتور قرار دهیم. به این ترتیب، اکتور تأیید تعداد واحد به محض ایجاد می‌تواند کار خود را شروع کند و اکتور اخذ درس نیازی به ارسال پیغام به آن ندارد. با توجه به این توضیحات پیغام پاسخ برای این شرط به صورت زیر می‌باشد:

UnitsValidationResponse(result:Boolean)

در این مرحله باید تعیین کنیم که اکتور بررسی اخذ درس با دریافت پاسخ هر پیغام چه عملی را باید انجام دهد:
هر یک از پاسخ‌هایی که اکتور بررسی اخذ درس دریافت می‌کند در واقع نتیجه‌ی بررسی یکی از شروط لازم برای اخذ

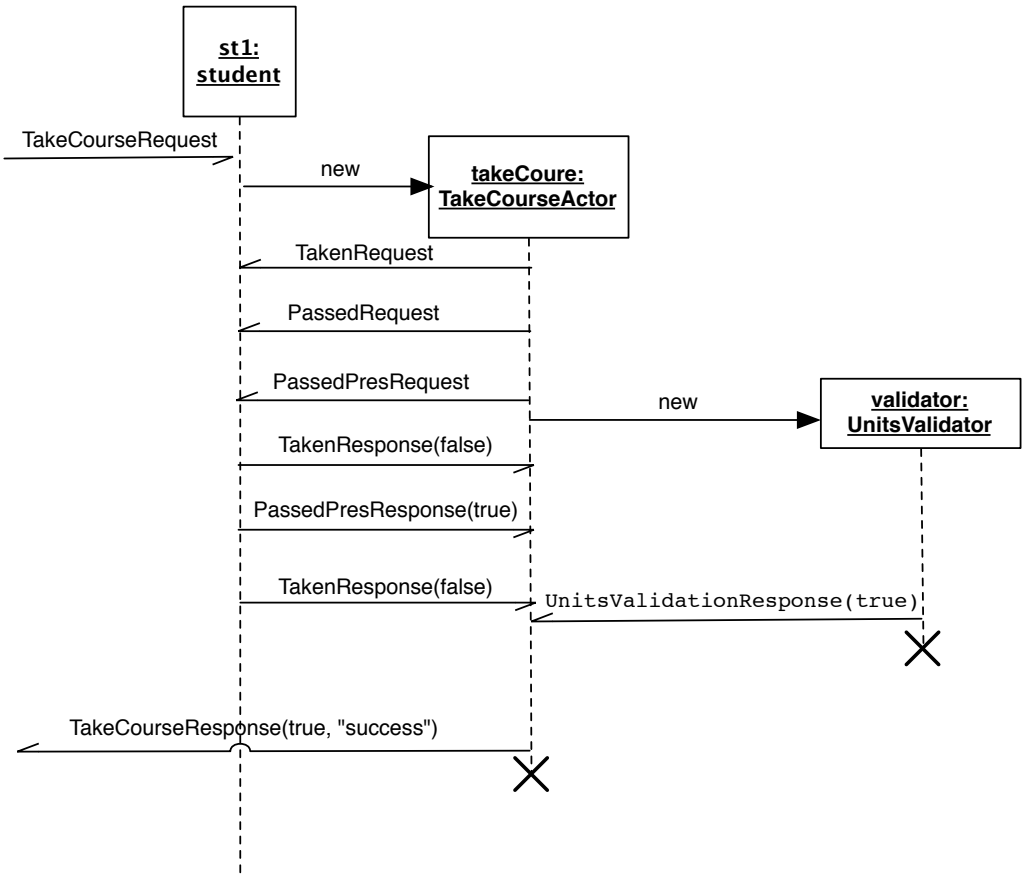
درس است. برای موافقت با اخذ درس توسط دانشجو، تمام شروط باید بررسی شوند. بنابراین پاسخ موافقت با اخذ درس فقط زمانی می‌تواند ارسال شود که تمام پاسخ‌ها دریافت شوند. برای اینکه اکتور اخذ درس از اتمام دریافت دروس مطلع شود لازم است که متغیری به منظور نگه‌داری تعداد پاسخ‌هایی که باید دریافت شود ایجاد شده بروزرسانی شود. با این کار اکتور اخذ درس می‌داند که چه زمانی کار به اتمام رسیده است. اما در این مورد کاربرد، در همه‌ی حالت‌ها لازم نیست اکتور منتظر تمام پاسخ‌ها بماند. دلیل این امر این است که در صورتی که هر یک از شروط اخذ درس نقض شود، نیازی به بررسی سایر شروط نیست. مثلاً اگر دانشجو قبلاً درس را گذرانده باشد نیازی به دریافت سایر پاسخ‌ها وجود ندارد و می‌توانیم پاسخ درخواست را ارسال کنیم (خطای گذرانده شدن درس). بنابراین در این مورد کاربرد با گرفتن هر پاسخ به این ترتیب عمل می‌کنیم که اگر شرط برقرار باشد، مقدار متغیر تعداد پاسخ‌های دریافت شده را یکی زیاد می‌کنیم، اگر مقدار جدید برابر با تعداد پاسخ مورد انتظار بود (این یعنی تمام پاسخ‌ها دریافت شده‌اند)، پاسخ نهایی درخواست را ارسال می‌کنیم. و اگر شرط نقض شده باشد پاسخ درخواست را که عدم موفقیت اخذ به دلیل نقض شرایط است ارسال می‌کنیم. نمودار شکل ۱۸.۴ تصمیمات اتخاذ شده تا این مرحله از طراحی را به صورت شماتیک نشان می‌دهد. در این نمودار حالتی بررسی شده که تمام شرایط اخذ درس برقرار شده و اخذ با موفقیت انجام می‌شود. حالت دیگری که یکی از شروط (تعداد واحد) برقرار نشده است در شکل ۱۹.۴ نشان داده شده است. در این حالت با توجه به اینکه یکی از پاسخ‌ها نشان‌دهنده‌ی این است که یکی از شروط برقرار نشده، به محض دریافت این پیغام، اکتور اخذ درس نتیجه‌ی درخواست را ارسال می‌کند و به کار خود پایان می‌دهد. طبیعتا پیغام‌های دیگری که برای این اکتور ارسال شده‌اند پردازش نخواهند شد. لازم به تأکید است که در هر دو شکل ترتیب پیغام‌ها فقط نشان دهنده‌ی یک حالت فرضی هستند. در عمل در هر بار اجرای برنامه، ترتیب گرفتن پاسخ‌ها ممکن است عوض شود.

در این مرحله، طراحی اکتور اخذ درس به پایان رسیده است. شبه‌کد ۲۰.۴ ساختار کلاس اکتور اخذ درس را نشان می‌دهد. در ادامه باید تغییرات سایر اکتورها در اثر دریافت پیغام‌های ارسال شده از اکتور اخذ درس اعمال شود و نیز اکتور جدیدی که ایجاد شده (اکتور تایید تعداد واحد) نیز طراحی گردد.

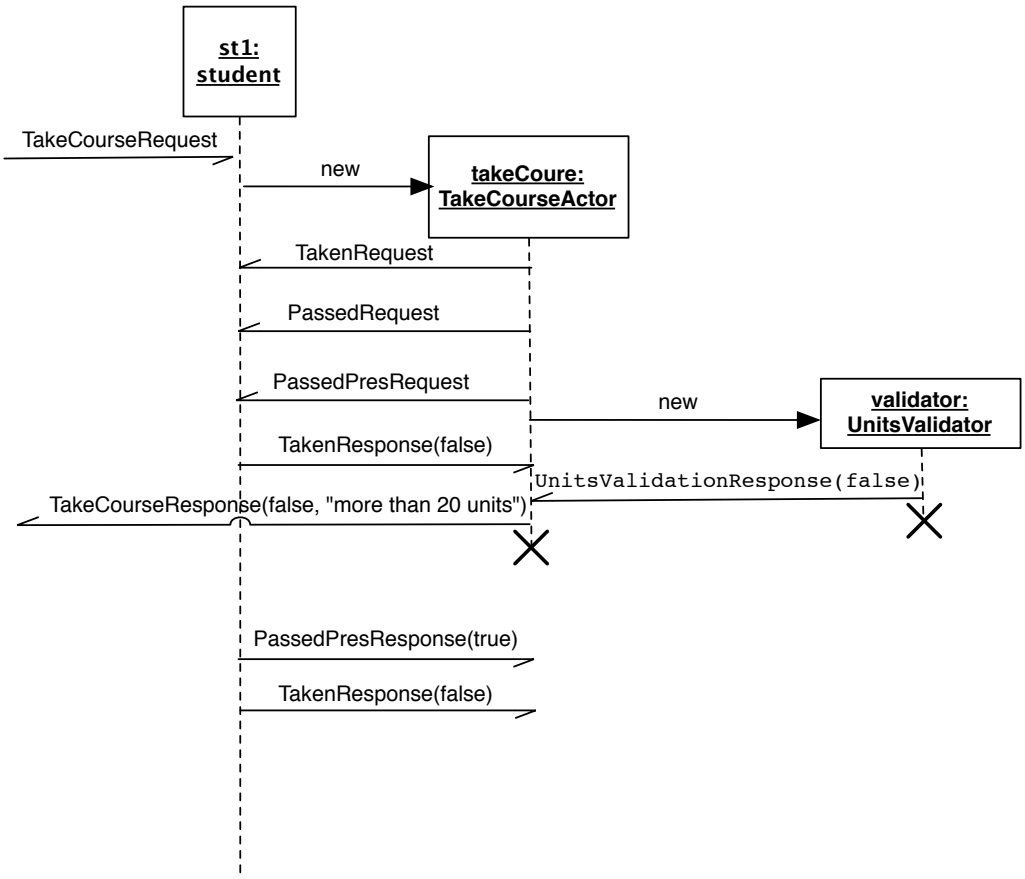
۲.۳.۳.۴ بررسی گذرانده شدن درس

برای بررسی گذرانده شدن درس، اکتور اخذ درس یک پیغام CoursePassRequest به اکتور دانشجو ارسال می‌کند. اکتور دانشجو بررسی این شرط را به اکتور جدید گذراندن درس[۳۱] می‌سپارد. این اکتور برای بررسی گذرانده شدن

[۳۱] CoursePassActor



شکل ۱۸.۴: نمودار ترتیب تبادل پیغام برای اخذ درس- حالتی که تمام شروط برای اخذ برقرار است



شکل ۱۹.۴: نمودار ترتیب تبادل پیغام برای اخذ درس- حالتی که یکی از شروط برقرار نیست

```
class StudentTakeCourseActor(
  val student: Student, course: Course, target: Actor) extends Actor {
  var receivedResponses:Int = 0
    override def act() {
      student ! PassedRequest(course)
      student ! TakenRequest(course)
      student ! PassedPresRequest(course)
      new UnitValidationActor(this, student, course).start
    loop { react {
        case PassedPresResponse(result) =>
          if(!result)
            sendResponse(false, "Student has not passed prerequisites")
          else waitForNextMessage()
        case PassedResponse(result) =>
          if(result)
            sendResponse(false, "Student has already passed this course")
          else waitForNextMessage()
        case TakenResponse(result) =>
          if(result)
            sendResponse(false, "Student has already taken this course")
          else waitForNextMessage()
        case UnitsValidationResponse(result) =>
          if(!result)
            sendResponse(false, "Student can't take more than 20 units")
          else waitForNextMessage()
      } }
  }
  def sendResponse(result: Boolean, comment: String) {
    target ! TakeCourseResponse(result, comment)
    exit
  }
  def waitForNextMessage() {
    receivedResponses++
    if(receivedResponses == 4) {
      //save a StudyRecord for course
      sendResponse(true, "successful")
    }
  }
}
```

شکل ۲۰.۴: شبه‌کد طراحی اکتور اخذ دانشجو.

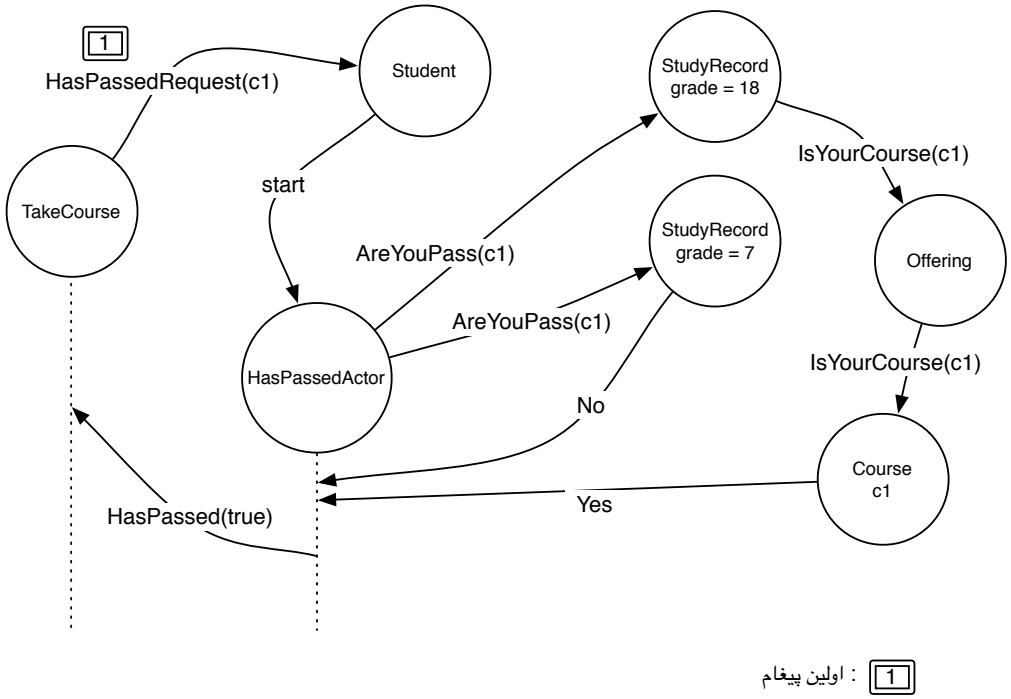
درس، نیاز به برقراری ارتباط با اکتورهای سابقه[۳۲] دارد. بنابراین اکتور دانشجو لیست سابقه‌ی دانشجو و نیز درسی که باید گذرانده شدن آن بررسی شود را در اختیار اکتور بررسی گذراندن درس قرار می‌دهد. اکتور گذراندن درس از تمام اکتورهای سابقه سؤال می‌کند که آیا سابقه‌ی مربوطه یک گذراندن موفق از درس مذکور است یا خیر. این کار با ارسال یک پیغام با قالب زیر صورت می‌پذیرد:

AreYouPassCourseRequest(course)

هر اکتور سابقه با دریافت این پیغام باید اولاً بررسی کند که آیا سابقه‌ای مربوط به درس مذکور است یا خیر، و ثانیاً نمره‌ی سابقه نمره‌ی قبولی است یا خیر. در اینجا با توجه به اینکه نمره‌ی مربوطه در اختیار خود اکتور سابقه است، بررسی آن ساده‌تر است. در صورتی که نمره کمتر از ۱۰ باشد، این اکتور بلافاصله پاسخ پیغام (منفی) را می‌دهد، در غیر این صورت برای بررسی این که این سابقه مربوط به درس مذکور است یا خیر، یک پیغام برای اکتور ارائه ارسال می‌کند. اکتور ارائه با گرفتن این پیغام آن را برای اکتور درس ارسال می‌کند تا این اکتور بررسی کند که آیا با درسی که در قالب پیغام دریافت کرده برابر است یا خیر. این اکتور پاسخ نهایی را مستقیماً برای اکتور بررسی گذرانده شدن درس ارسال می‌کند.

شکل ۲۱.۴ تبادل پیغام‌های مربوط به بررسی گذرانده شدن درس را به صورت شماتیک نشان می‌دهد. در این شکل خط عمودی در حالتی به کار رفته است که یک اکتور در زمان‌های مختلف پیغام دریافت کرده باشد. در مثال بررسی شده در شکل، اکتور اخذ درس یک درخواست بررسی گذرانده شدن درس (درس c۱ (را برای اکتور دانشجو ارسال می‌کند. اکتور دانشجو یک اکتور بررسی گذرانده شدن درس (HasPassedActor) ایجاد می‌کند و کار بررسی را به آن واگذار می‌کند. این اکتور پیغام‌های مناسب را برای دو اکتور سابقه‌ی دانشجو ارسال می‌کند. یکی از اکتورهای سابقه به این دلیل که نمره‌ی کمتر از ۱۰ (۷) دارد، بلافاصله پاسخ را برای اکتور مقصد ارسال می‌کند. اکتور سابقه‌ی دیگر با توجه به اینکه نمره‌ی قبولی دارد، برای اطمینان از اینکه مربوط به همان درسی است که گذرانده شدن آن بررسی می‌شود، یک پیغام به اکتور ارائه ارسال می‌کند. اکتور ارائه پیغام را به درس منتقل می‌کند و اکتور درس با مقایسه‌ی درس موجود در پیغام با خودش، جواب را برای مقصد می‌فرستد.

[۳۲] StudyRecord

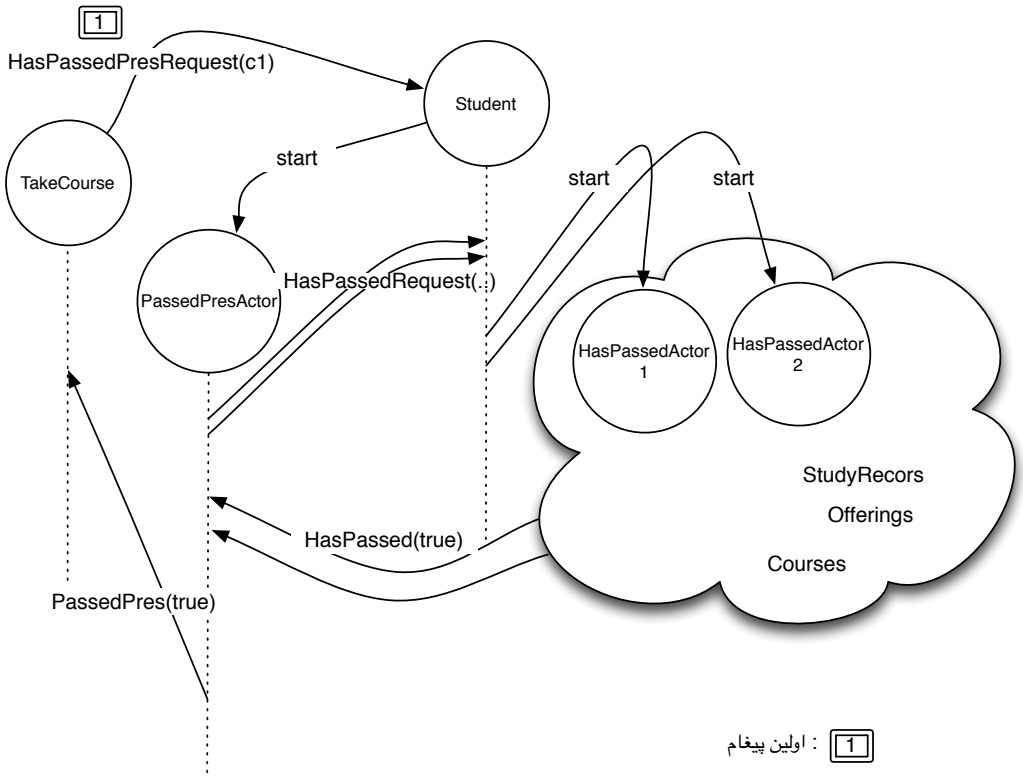


شکل ۲۱.۴: نمایش شماتیک تبادل پیغام بین اکتورهای مختلف برای بررسی گذرانده شدن یک درس

۳.۳.۳.۴ بررسی گذرانده شدن پیش‌نیازهای درس

اکتور دانشجو با دریافت پیغام بررسی گذرانده شدن پیش‌نیازهای درس، محاسبات مربوطه را به اکتوری که به این منظور طراحی شده ارسال می‌کند. با توجه به اینکه پیش‌نیازهای هر درس نیز خود از نوع درس هستند، در طراحی این بخش می‌توان از اکتور بررسی گذرانده شدن درس استفاده کرد. بنابراین اکتور مذکور به ازای هر کدام از پیش‌نیازهای درس، یک پیغام بررسی گذرانده شدن درس به دانشجو ارسال می‌کند. با دریافت هر پاسخ اگر مشخص شود که درسی از میان پیش‌نیازها گذرانده نشده است، بلافاصله پاسخ درخواست به مقصد (اکتور اخذ درس) ارسال می‌شود. در غیر این صورت پس از گرفتن تمام پاسخ‌ها، یک پیغام به اکتور اخذ درس ارسال می‌کند و به وسیله‌ی آن اعلام می‌کند که تمام پیش‌نیازها گذرانده شده است. شکل ۲۲.۴ ارتباط اکتورها برای بررسی گذرانده شدن درس نشان می‌دهد. در این شکل، ابتدا اکتور اخذ درس پیغام بررسی گذرانده شدن درس c۱ را به اکتور دانشجو ارسال می‌کند. اکتور دانشجو بررسی بررسی این مورد را به اکتور بررسی پیش‌نیاز[۳۳] واگذار می‌کند. این اکتور به ازای هر کدام از پیشنیازهای درس، یک پیغام بررسی گذرانده شدن درس برای خود دانشجو ارسال می‌کند. طراحی مورد بررسی گذراندن درس در بخش ۲.۳.۳.۴

[۳۳] PassPresActor



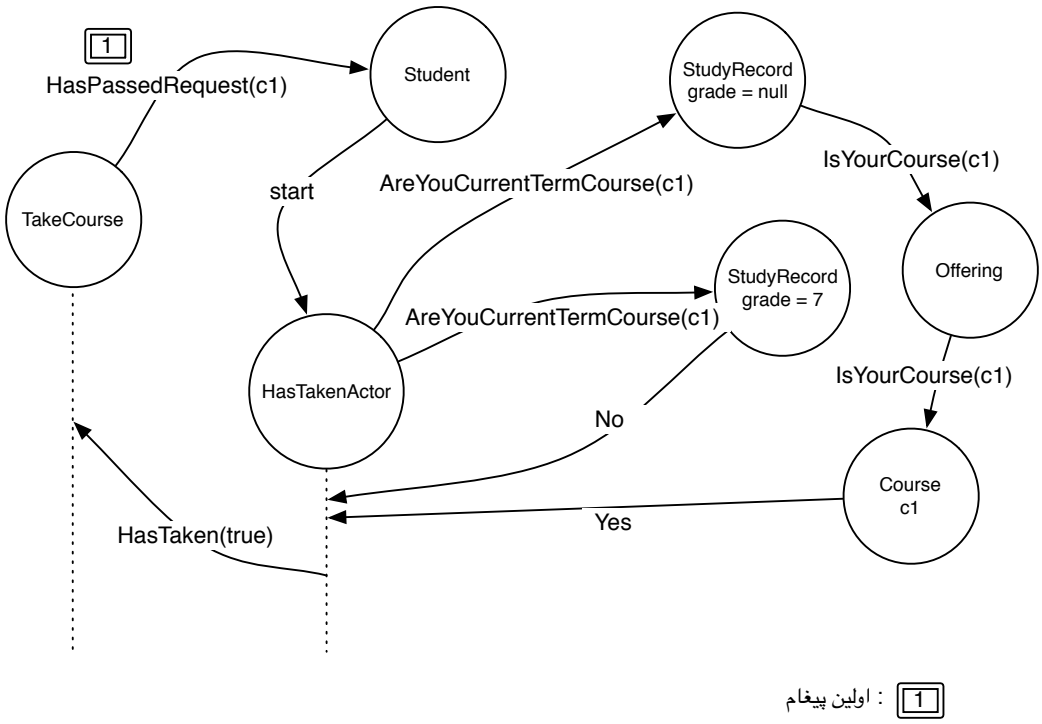
شکل ۲۲.۴: نمایش شماتیک تبادل پیغام بین اکتورهای مختلف برای بررسی گذرانده شدن پیشنیازهای یک درس

توضیح داده شد. در این شکل برای جلوگیری از پیچیدگی طراحی، از نمایش نحوه‌ی بررسی گذرانده شدن درس صرف نظر شده است و اکتورها و پیغام‌های مربوط به آن به صورت شکل ابر نمایش داده شده است.

۴.۳.۳.۴ بررسی عدم اخذ مجدد درس

همان‌طور که در بخش ۱.۳.۳.۴ توضیح داده شد، اکتور اخذ درس یک پیغام برای بررسی عدم اخذ مجدد درس برای اکتور دانشجو ارسال می‌کند. اکتور دانشجو مطابق حالت‌های قبل بررسی این مورد را به اکتور بررسی اخذ درس[۳۴] واگذار می‌کند. بررسی اخذ شدن درس کاملاً مشابه بررسی گذرانده شدن درس است. در بررسی گذرانده شدن درس، اکتور سابقه نمره را بررسی می‌کند، در صورتی که نمره قبولی نباشد جواب را ارسال می‌کند و در صورتی که نمره قبولی باشد برای بررسی اینکه درس مربوط به سابقه همان درس مورد سؤال است یا خیر، با اکتور ارائه تبادل پیغام انجام می‌دهد. در بررسی عدم اخذ مجدد درس، از هر اکتور سابقه سؤال می‌شود که آیا سابقه مربوط به ترم جاری است یا خیر. برای اینکه یک سابقه مربوط به ترم جاری باشد، کافی است نمره‌ای برای آن اعلام ثبت نشده باشد. بنابراین اکتور سابقه بررسی می‌کند که نمره‌ای برایش ثبت شده یا خیر اگر مقدار فیلد نمره null باشد یعنی مربوط به ترم جاری است و برای بررسی اینکه مربوط به همان درس مورد سؤال است مانند حالت بررسی گذرانده شدن درس، یک پیغام به ارائه ارسال می‌کند. در غیر این صورت حتماً جواب منفی است و بلافاصله یک پیغام برای اکتور بررسی عدم اخذ مجدد ارسال می‌شود. شکل ۲۳.۴ تبادل پیغام بین اکتورها برای بررسی این شرط را نشان می‌دهد. همان‌طور که مشاهده می‌شود این شکل بسیار شبیه به شکل ۲۱.۴ است که بررسی گذرانده شدن درس را نشان می‌دهد.

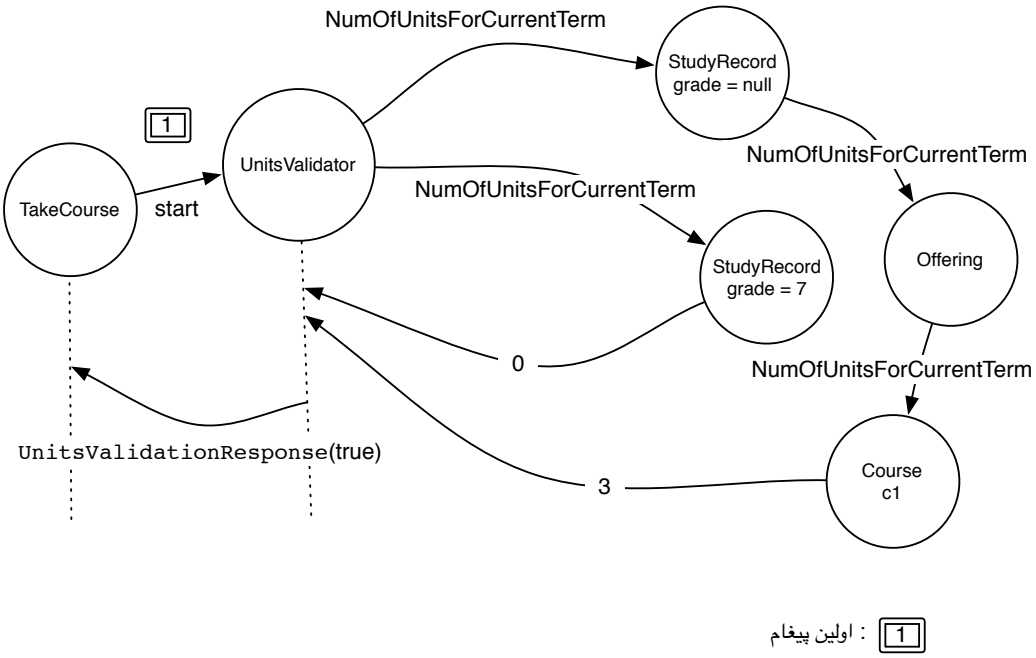
[۳۴] CourseTakenCheckActor



شکل ۲۳.۴: نمایش شماتیک تبادل پیغام بین اکتورهای مختلف برای بررسی عدم اخذ مجدد درس

۵.۳.۳.۴ بررسی عدم اخذ بیش از ۲۰ واحد

در بخش طراحی اکتور اخذ درس (بخش ۱.۳.۳.۴) دیدیم که اکتور اخذ درس برای بررسی این شرط که تعداد واحدهای اخذ شده بیشتر از ۲۰ نشود، یک اکتور به این منظور ایجاد می‌کند. این اکتور با بررسی این شرط نتیجه را به صورت پیغام UnitsValidationResponse برای اکتور اخذ درس ارسال می‌کند. طراحی این کارکرد به این صورت است که اکتور مورد نظر از تمام سابقه‌های ترم درخواست می‌کند تا در صورتی که مربوط به ترم جاری هستند، تعداد واحدهای درس مربوط به خود را ارسال کنند. همان‌طور که در بخش قبل توضیح داده شد، اینکه سابقه مربوط به ترم جاری است از null بودن فیلد نمره مشخص می‌شود. بنابراین اگر فیلد نمره null نباشد اکتور سابقه عدد صفر را به عنوان تعداد واحد ارسال می‌کند. در غیر این صورت مشابه حالت‌های قبل یک پیغام برای اکتور درس (از طریق اکتور ارائه) ارسال می‌کند تا تعداد واحدها را به اکتور مقصد (اکتور بررسی عدم اخذ بیش از ۲۰ واحد) ارسال کند. اکتور مذکور با گرفتن هر پیغام تعداد واحدها را بروزرسانی می‌کند و با اتمام پیغام‌ها بررسی می‌کند که آیا جمع واحدها بیش از ۲۰ است یا خیر. در نهایت پاسخ را برای اکتور اخذ درس ارسال می‌کند. شکل ۲۴.۴ همکاری اکتورها برای بررسی این شرط را به صورت شماتیک نشان می‌دهد.



شکل ۲۴.۴: نمایش شماتیک تبادل پیغام بین اکتورهای مختلف برای بررسی عدم اخذ بیش از ۲۰ واحد

فصل ۵

روش طراحی و الگوها

در بخش‌های پیشین یک سیستم نمونه معرفی شد و پس از توصیف موارد کاربرد آن، روش طراحی آن با استفاده از مدل تبادل ناهمگام پیغام بررسی شد. در ادامه‌ی این فصل تلاش می‌شود با توجه به تجربیات حاصل از انجام این طراحی، روش معرفی شده به صورت نظام‌مند معرفی شود. در قسمت اول از این بخش، قدم‌های لازم برای طراحی یک سیستم به روش تبادل ناهمگام پیغام ذکر شده و در موارد ممکن، از قسمت‌هایی از سیستم طراحی شده به عنوان نمونه بهره گرفته شده است. در قسمت بعد تلاش شده الگوهای کلی هماهنگی اکتورها با تمرکز بر خواص منطق دامنه بررسی شود. در بخش بعد قسمتی از تجربیات حاصل از بررسی رویکردهای متعدد برای طراحی سیستم نمونه (سیستم آموزش ساده) ارائه شده است. و نهایتا در بخش پایانی قسمتی بحث مختصری در مورد نکات برنامه‌نویسی در هنگام پیاده‌سازی سیستم مطرح شده است.

۱.۵ گام‌های طراحی به روش تبادل ناهمگام پیغام

روش بررسی شده در این پژوهش ارتباط تنگاتنگی با مبحث طراحی شیءگرا دارد. طراحی شیءگرا با استفاده از لفافه‌بندی[۱] اشیاء منجر به تفکیک واسط کارکردی یک شیء از حالت محلی آن می‌شود و جزئیات پیاده‌سازی رفتار را مخفی می‌کند. این خاصیت منجر به افزایش امکان استدلال در مورد نحوه‌ی طراحی اشیاء می‌شود. در این روش، مکانیزم کنترل اجرای

[۱] encapsulation

برنامه‌ها فراخوانی متد است. روش تبادل ناهمگام به تفکیک کنترل اجرای منطق برنامه از زمان اجرای آن می‌پردازد. به این ترتیب قابلیت افزودن همروندی در طراحی را اضافه می‌کند. آلن کِی در [۳۱] اظهار داشته است که ایده‌ی اصلی در طراحی شیءگرا، ارسال پیغام بوده است. و این مسئله ارتباط تنگاتنگ طراحی شیءگرا و طراحی مبتنی بر تبادل ناهمگام پیغام را نشان می‌دهد.

بنابراین بسیاری از ایده‌های طراحی و تحلیل شیءگرا عیناً در این روش نیز کاربرد دارند. به همین دلیل در ارائه‌ی روش طراحی به بررسی جزئیات مواردی که دقیقاً مشابه طراحی شیءگرا هستند پرداخته نشده است. علاوه بر این، در ارائه‌ی روش فرض شده که خروجی‌های تحلیل سیستم موجود هستند. طبیعتاً روش‌های تحلیل شیءگرا و کسب شناخت از سیستم تحت طراحی، عیناً قابل اعمال در این نوع طراحی هستند. در گام‌های ذکر شده برای طراحی به روش تبادل ناهمگام، بعضی از گامها مربوط به خروجی‌های تحلیل سیستم هستند که برای حفظ انسجام، توضیح داده شده‌اند. با توجه به این موارد، در این بخش گام‌های طراحی به روش تبادل پیغام را ارائه می‌کنیم:

۱.۱.۵ شناخت سیستم و تشخیص اکتورهای دامنه

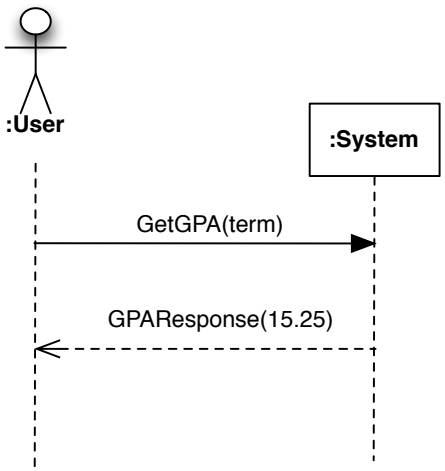
شناخت سیستمی که باید طراحی شود پیش از شروع به طراحی لازم است. فعالیت‌های مربوط به این بخش مشابه همین فعالیت‌ها در روش‌های تحلیل نیازمندی‌ها و کسب شناخت[۲] در متدولوژی‌های طراحی شیءگرا است و جزئیات آنها در حوزه‌ی این پژوهش نمی‌باشد. تعدادی از خروجی‌های این فعالیت‌ها از جمله توصیف موارد کاربرد سیستم و استخراج اشیاء دامنه به طور گسترده در طراحی مورد استفاده قرار می‌گیرند. در مدل اکتور، همه‌ی موجودیت‌های سیستم اکتور هستند. بنابراین تمام اشیاء مدل دامنه‌ی سیستم که در مراحل ابتدایی طراحی و تحلیل شناسایی می‌شوند، به صورت اکتور طراحی می‌شوند. در منابع تحلیل و طراحی شیءگرا، روش‌هایی برای تشخیص اشیاء دامنه و نمایش مناسب آنها بیان شده است که طبیعتاً قابل اعمال در این روش نیز می‌باشند[۳۲]. در سیستم آموزش معرفی شده، مدل دامنه در قالب نمودار کلاس در بخش ۲.۲.۴ نمایش داده شده است.

۲.۱.۵ انتخاب مورد کاربرد برای طراحی جزئیات

با در دست داشتن موارد کاربرد و اشیاء دامنه، فعالیت‌های مربوط به طراحی اکتورهای سیستم آغاز می‌گردد. در گام اول نیاز داریم یکی از موارد کاربرد را برای طراحی انتخاب کنیم. معمولاً انتخاب مورد کاربرد با توجه به اولویت و اهمیت

[۲] Inception

آن صورت می‌پذیرد. پس از انتخاب مورد کابرد باید رخدادهای سیستمی آن شناسایی شوند. این رخدادها نتیجه‌ی تعامل بازیگران خارجی با سیستم هستند. استخراج رخدادهای سیستمی با توجه به موارد کاربرد صورت می‌گیرد. این رخدادها را می‌توان با استفاده از نمودارهای ترتیب سیستمی[۳] نمایش داد[۳۲]. در نمودار ترتیب سیستمی، سیستم به صورت جعبه‌ی سیاه[۴] درنظر گرفته می‌شود و تعامل بازیگر خارجی با سیستم به صورت فرستادن درخواست و دریافت پاسخ نمایش داده می‌شود. به عنوان مثال شکل ۱.۵ نمودار ترتیب سیستمی را برای سناریوی اصلی مورد کاربرد محاسبه‌ی معدل نشان می‌دهد. رخدادهای سیستمی نقطه‌ی مناسبی برای شروع به طراحی اکتورها و ارتباطات آنها هستند. در طراحی شیءگرای ترتیبی، رخدادهای سیستمی در ارتباطی تنگاتنگ با متدهای یک شیء قرار دارند. در واقع نقطه‌ی آغاز اجرای محاسبات مربوط به یک رخداد سیستمی یک متد است. دلیل این پیش‌فرض این است که مکانیزم کنترل برنامه در طراح شیءگرای ترتیبی (و هر روش ترتیبی دیگر) فراخوانی متد است. در حالی که در روش مبتنی بر اکتور، مکانیزم ارتباطی تبادل پیغام است. بنابراین در این روش هر رخداد سیسستمی به یک پیغام نگاشت می‌شود که به دست یکی از اکتورها سیستم می‌رسد.



شکل ۱.۵: نمودار ترتیب سیستمی برای یک سناریو از مورد کاربرد محاسبه‌ی معدل

[۳] system sequence diagram (SSD)

[۴] black box

۳.۱.۵ طراحی اکتور اول

همان‌طور که در مورد قبل توضیح داده شد، در مدل طراحی اکتور، وقوع یک رخداد سیستمی، به وسیله‌ی ارسال پیغام صورت می‌گیرد. گام اول در طراحی سیستم به هدف پاسخگویی به این پیغام این است که مشخص شود کدام اکتور باید اولین پیغام را دریافت کند. این مورد در طراحی شیءگرا در قالب مفهوم **مسئولیت شیء** بیان می‌شود. در طراحی شیءگرا شیءای موظف به دریافت درخواست است که مسئولیت درخواست با آن باشد. تشخیص مسئولیت با توجه به منطق دامنه صورت می‌گیرد و بدون در نظر گرفتن منطق دامنه، قاعده‌ای برای انتخاب شیء مسئول وجود ندارد. مسئولیت اشیاء از نظر نوع به دو دسته‌ی کلی مسئولیت *انجام*[۵] (مانند مسئولیت ایجاد یک شیء و آغاز یک عملیات) و مسئولیت *اطلاع*[۶] (مانند اطلاع از اشیاء مرتبط یا اطلاع از داده‌های محلی لفافه‌بندی شده) تقسیم می‌شود[۳۲، ۳۳]. معمولاً مسئولیت پاسخگویی به درخواست‌های سیستمی بعد از تهیه‌ی مدل دامنه آسان‌تر می‌شود. در سیستم بررسی شده در این پژوهش مسئولیت پاسخ‌گویی به درخواست محاسبه‌ی معدل اکتور دانشجو است. با مشخص شدن اکتور مسئول برای دریافت پیغام، در ادامه‌ی طراحی باید روش پردازش پیغام در اکتور مورد نظر بررسی گردد.

۴.۱.۵ منطق پردازش درخواست

تا این مرحله از طراحی، مشخص شده است که اکتور مسئول برای دریافت پیغام درخواست کدام است. در این مرحله باید تصمیم گرفته شود که نحوه‌ی پردازش پیغام درخواست به چه صورتی خواهد بود.

۱.۴.۱.۵ تصمیم‌گیری برای انتقال پردازش درخواست به اکتور دیگر

پردازش پیغام به دو صورت کلی انجام می‌پذیرد. حالت اول این است که خود اکتور مسئولیت پردازش پیغام را بر عهده بگیرد. در این حالت اکتور مذکور یا به تنهایی قادر به انجام تمام عملیات مرتبط با درخواست دریافت شده است و یا با همکاری اکتورهای دیگر می‌تواند منطق مربوط به درخواست را اجرا کند.

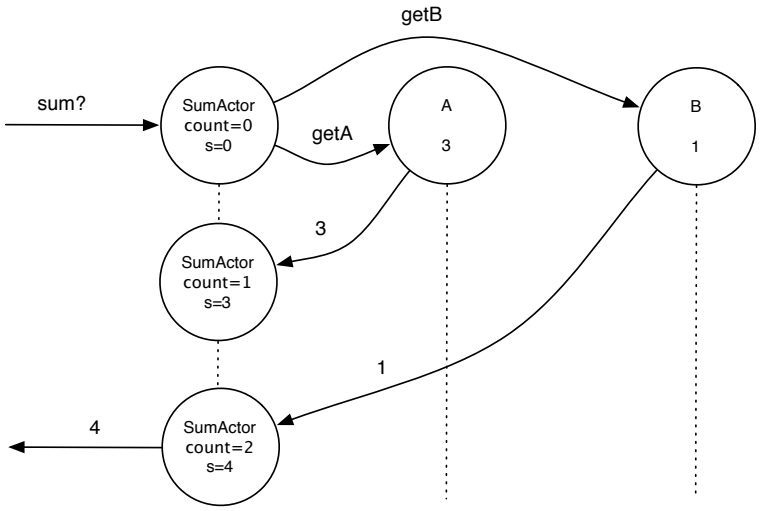
حالت دوم به این صورت است که اکتور تصمیم بگیرد که اکتور جدیدی را به منظور پردازش این درخواست ایجاد کند و تمام عملیات مربوط به درخواست را به این اکتور واگذار کند. حالت مشابه این مورد در طراحی شیءگرای ترتیبی

[۵] Doing Responsibilities

[۶] Knowin Responsibilities

نیز رخ می‌دهد. در طراحی شیءگرا ممکن است به دلیل جلوگیری از افزایش پیچیدگی کلاس و حفظ قابلیت تغییر، تمام کار پردازش یک درخواست را به کلاس دیگری که به همین منظور ایجاد می‌شود منتقل کند. در روش طراحی مبتنی بر تبادل ناهمگام، علاوه بر این مورد به دلیل دیگری نیز این تصمیم اتخاذ می‌شود. این مسئله در ادامه در قالب یک مثال توضیح داده می‌شود:

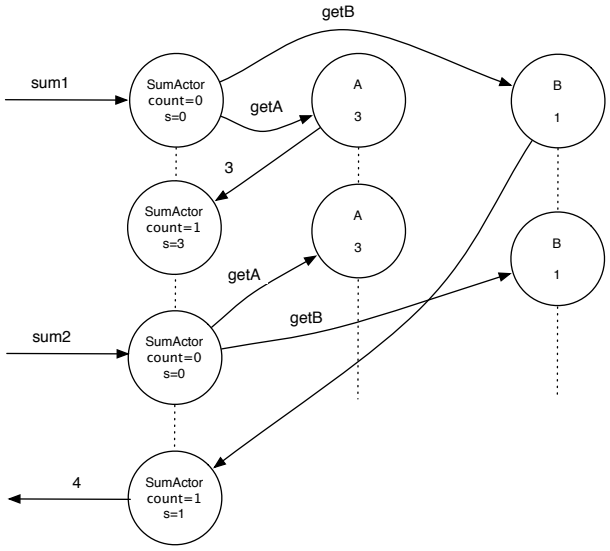
فرض کنید یک مدل دامنه از ۳ اکتور تشکیل شده باشد. اکتور A که یک متغیر محلی عددی به نام a دارد. این اکتور یک نوع پیغام دریافت می‌کند: پیغام getA که در پاسخ آن مقدار a را ارسال می‌کند. اکتور B به طور مشابه یک متغیر محلی عددی به نام b دارد و با دریافت پیغام getB مقدار b را ارسال می‌کند. اکتور سوم Sum نام دارد که با دریافت پیغام sum باید مجموع مقادیر a و b را ارسال کند. اکتور Sum چون از مجموع a و b اطلاع ندارد برای پاسخ به پیغام sum نیاز به همکاری A و B دارد. فرض کنیم اکتور Sum به این شکل طراحی می‌شود که با دریافت پیغام sum ابتدا متغیرهای محلی s و count را صفر می‌کند و سپس پیغام‌های getA و getB را برای اکتورهای A و B ارسال می‌کند با دریافت هر پاسخ مقدار count را یکی زیاد می‌کند و متغیر s را با عدد دریافت شده جمع می‌کند. بعد از دریافت هر پاسخ و بروزرسانی متغیرهای داخلی، اگر مقدار count برابر با عدد ۲ شد (یعنی هر دو پاسخ دریافت شده است) مقدار متغیر s (حاصل‌جمع) را به عنوان پاسخ درخواست ارسال می‌کند. شکل ۲.۵ همکای این ۳ اکتور برای پاسخ به درخواست sum را نشان می‌دهد. مشکل این طراحی زمانی مشخص می‌شود که اکتور Sum بعد از دریافت پاسخ A



شکل ۲.۵: همکاری موفق اکتورها برای پاسخ به درخواست مجموع a و b

و قبل از دریافت پاسخ B یک درخواست sum دیگر دریافت می‌کند. اکتور Sum برای پاسخ به این درخواست مقدار متغیرهای s و count را صفر می‌کند و دو پیغام جدید برای A و B ارسال می‌کند. در این هنگام اکتور Sum پاسخ

B برای درخواست اول را دریافت می‌کند اما چون مقدار متغیر count صفر است، متوجه اتمام عملیات درخواست اول نمی‌شود. شکل ۳.۵ این حالت را نمایش می‌دهد. مثال فوق نشان می‌دهد که در طراحی به روش تبادل ناهمگام



شکل ۳.۵: مشکل پیغام‌های همروند در همکاری اکتورها برای پاسخ به درخواست مجموع a و b

پیغام در صورتی که نیاز به همکاری بین اکتورها وجود داشته باشد ممکن است درخواست‌های همروند موجب تداخل در محاسبات همدیگر بشوند. این مشکل زمانی پیش می‌آید که شرایط زیر برقرار باشند:

۱. برای پاسخ به یک درخواست، اکتور مجبور به همکاری با سایر اکتورها باشد.

۲. همکاری به ارسال پیغام به سایر اکتورها ختم نشود و در ادامه لازم باشد پاسخ آنها دریافت شود.

۳. ارسال پیغام‌ها به صورت ناهمگام انجام شود. به این معنی که اکتور مورد نظر پس از ارسال پیغامهای مربوطه بتواند بدون توقف برای دریافت پاسخ‌ها به پردازش سایر پیغام‌ها بپردازد.

۴. اکتور با دریافت هر کدام از پاسخ‌ها متغیر(های) محلی خود را بروزرسانی کند.

راه حل مشکل:

برای حل این مشکل در زمان طراحی چند گزینه پیش رو داریم:

- گزینه‌ی اول این است که به جای ارسال ناهمگام پیغام‌های مربوط به یک درخواست، عمل تبادل پیغام را به صورت همگام انجام دهیم. در این حالت اطمینان حاصل می‌شود که درخواست جدید قبل از اتمام عملیات درخواست

قبلی پردازش نمی‌شود. ایراد این روش این است که باعث محدودیت در همروندی سیستم می‌گردد.

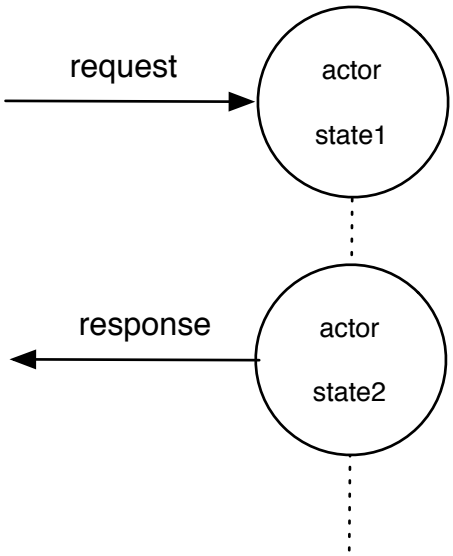
- راه دوم این است که به گونه‌ای متغیرهای حالت را نگهداری کنیم که با پردازش درخواست‌های همروند دچار مشکل نشوند. برای این کار نیاز داریم تا متغیرهای حالت را برای درخواست‌های پردازش شده اختصاصی کنیم. در مثال معرفی شده در شکل ۲.۵ این کار به این صورت انجام می‌شود که به جای متغیرهای s و count برای هر درخواست متغیر جدیدی در نظر بگیریم. طبیعتاً این روش منجر به ایجاد پیچیدگی در زمان پیاده‌سازی می‌شود و تضمین محافظت از متغیرها کار دشواری خواهد بود.

- راه مناسب برای حل این مشکل این است که تمام عملیات مربوط به درخواست دریافت شده، به یک اکتور دیگر که فقط وظیفه‌ی پاسخ به این درخواست را دارد منتقل شود. به این روش هم مشکل متغیرهای حالت محلی به وجود نمی‌آید و هم همروندی سیستم محدود نمی‌شود. طبیعتاً برای جلوگیری از تکرار همین مشکل برای اکتور جدید باید به ازای هر درخواست مشابه یک نمونه‌ی جدید از اکتور مذکور ایجاد کنیم.

  بنابراین در طراحی به روش تبادل ناهمگام پیغام علاوه بر جلوگیری از پیچیدگی و بزرگ شدن بیش از حد کلاس، ممکن است به دلیل ایجاد امکان پردازش درخواست‌های همروند نیز تصمیم به واگذاری درخواست به اکتوری جدید نماییم. در طراحی سیستم آموزش که در بخش‌های قبل توضیح داده شد، در موارد متعددی از جمله در مورد کاربرد محاسبه‌ی معدل (بخش ۲.۲.۳.۴) این الگو مشاهده شد.

۲.۴.۱.۵ پردازش پیغام بدون همکاری با سایر اکتورها

مستقل از این موضوع که پردازش یک درخواست توسط خود اکتور انجام می‌شود یا به اکتور جدیدی منتقل می‌شود، در طراحی سیستم باید مشخص شود که منطق مربوط به یک درخواست چگونه پیاده‌سازی خواهد شد. ساده‌ترین حالت برای اجرای منطق مربوط به یک درخواست این است که اکتور دریافت کننده، بتواند بدون همکاری با سایر اکتورها عملیات لازم را انجام دهد و در صورت لزوم پاسخ درخواست را ارسال کند. این حالت زمانی رخ می‌دهد که اکتور تمام اطلاعات لازم برای اجرای منطق مربوطه را در حالت خود داشته باشد. در این حالت اکتور پس از دریافت پیغام، منطق مربوط به آن را انجام می‌دهد و در صورت لزوم پاسخ مناسب را برای مقصد ارسال می‌کند. ممکن است حالت اکتور پس از پردازش این پیغام تغییر کند. شکل ۴.۵ این حالت را نشان می‌دهد.



شکل ۴.۵: پردازش پیغام بدون همکاری با سایر اکتورها

۳.۴.۱.۵ پردازش پیغام به وسیله‌ی همکاری با سایر اکتورها

در بخش قبل ذکر شد که ممکن است یک اکتور با دریافت یک پیغام درخواست، قادر باشد بدون برقراری ارتباط با سایر اکتورها پاسخ درخواست را بدهد. در مواردی که اطلاعات مورد نیاز برای پردازش درخواست را در اختیار نداشته باشد، باید با اکتورهای دیگر ارتباط برقرار کند. برای برقراری ارتباط با سایر اکتورها از تبادل پیغام استفاده می‌شود. در برقراری ارتباط با سایر اکتورها نکات زیر باید در نظر گرفته شوند:

- بررسی اینکه تبادل پیغام به صورت همگام صورت گیرد یا ناهمگام:

یک عامل مهم در این تصمیم‌گیری منطق مربوط به پردازش پیغام است. در برخی موارد گزینه‌ای جز همگام‌سازی پیغام‌ها وجود ندارد.مثلاً اگر منطق مورد نظر احتیاج به پردازش به ترتیب زمانی داشته باشد و برای هر بخش آن نیاز به برقراری ارتباط با سایر اکتورها وجود داشته باشد، ارسال پیغام باید به صورت همگام صورت پذیرد. یا در حالتی که از اطلاعات مورد نیاز برای یک مرحله از پردازش پیغام، وابسته به نتیجه‌ی مرحله‌ی قبل باشد. در چنین مواردی ملزم به استفاده از تبادل همگام پیغام هستیم. البته این به این مفهوم نیست که فقط در این حالت می‌توان از پیغام همگام استفاده کرد. در واقع استفاده از ارتباط همگام، هیچ محدودیت ذاتی در مدل اکتور ندارد. حتی در این روش می‌توان یک مدل ترتیبی را با نگاشت هر متد از شیء به یک ارتباط همگام، به صورت مدل اکتور طراحی

کرد. در این حالت با اینکه هر اکتور به صورت همروند اجرا می‌شود، در اجرای برنامه همروندی وجود نخواهد داشت. بنابراین از ارتباط ناهمگام برای ایجاد همروندی در پردازش درخواست‌ها استفاده می‌شود.

- بررسی اینکه پیغام پاسخ در چه زمانی و توسط چه اکتوری ارسال شود:

پردازش یک پیغام ممکن است نیازی به ارسال پاسخ نداشته باشد. این نوع پیغام‌ها ممکن است صرفاً برای ایجاد تغییر در سیستم ارسال شوند و فرستنده نیازی به اطلاع از نتیجه نداشته باشد. در این حالت پس از پردازش پیغام، عملیات مربوط به آن تمام می‌شود. اما در بیشتر موارد ارسال کننده‌ی درخواست نیاز به اطلاع از نتیجه‌ی کار دارد و یا اطلاعاتی را درخواست کرده که باید به صورت پاسخ دریافت کند. در این موارد باید یک پاسخ هم برای فرستنده‌ی درخواست ارسال شود. یک تفاوت مهم مدل اکتور با مدل طراحی شیءگرای ترتیبی، در این است که در مدل مدل ترتیبی، پاسخ یک درخواست الزاماً توسط همان شیءای داده می‌شود که درخواست به آن داده شده است. البته باید دقت شود که در اینجا منظور این نیست که کل پردازش درخواست را شیء مذکور انجام می‌دهد. بلکه منظور این است که نهایتاً نتیجه‌ی کار از همان متدی بازگشت داده می‌شود که فراخوانی شده است. اما در مدل اکتور به دلیل همروندی سیستم، پیغام پاسخ می‌تواند توسط هر کدام از اکتورهای همکاری کننده ارسال شود. بنابراین در طراحی به این روش باید دقت کرد که طراحی را مقید به این نکنیم که دریافت کننده‌ی یک پیغام حتماً پاسخ آن را بدهد. در رویکرد دوم طراحی مورد کاربرد معدل (بخش ۲.۲.۳.۴) به طور مفصل به این مورد پرداخته شده است.

- استفاده از الگوهای همکاری اکتورها:

برای اینکه چند اکتور بتوانند باهم همکاری کرده و پاسخ یک درخواست را ارسال کنند راهکارهای طراحی متعددی وجود دارد که برخی از آنها در طراحی سیستم آموزش در بخش‌های قبل ذکر شده است. برخی از این روش‌ها به دلیل تکرار در حالت‌های زیاد و قابلیت استفاده‌ی مجدد، حالت الگو به خود می‌گیرند. در بخش‌های بعد تعدادی از این الگوها معرفی شده است.

- انتخاب اکتورهای مقصد برای پیغام‌های مربوط به درخواست

در صورتی که تصمیم به همکاری بین اکتورها گرفته شده باشد، پس از طراحی مدل همکاری (با استفاده از الگوهای همکاری اکتورها)، باید تبادل پیغام بین اکتورهایی که همکاری می‌کنند، طراحی شود. گام اول در این طراحی انتخاب مقصد پیغام است. این کار با بررسی مسئولیت‌های اکتورها با توجه به منطق دامنه انجام می‌شود. در صورتی که هیچ یک از اکتورهایی که تا این مرحله طراحی شده‌اند مسئول پردازش پیغام تشخیص داده نشوند، باید اکتور جدیدی برای پردازش پیغام طراحی شود.

- طراحی قالب پیغام‌های مربوط به درخواست

در این مرحله از طراحی باید پیغام‌های مربوط به منطق یک درخواست طراحی شده و ارسال شوند. در طراحی یک پیغام نکات زیر باید مورد توجه قرار گیرد:

- نام پیغام متناسب با عملکرد آن در نظر گرفته می‌شود. به طور کلی در طراحی شیءگرا توصیه می‌شود که نام کلاس‌ها و متدها متناسب با عملکرد و مسئولیت کلاس در نظر گرفته شود.
- اطلاعات مورد نیاز سایر اکتورهایی که پیغام به دست آنها خواهد رسید در پیغام قرار داده شود. اطلاعات قرار داده شده در پیغام به چند دسته تقسیم می‌شوند:

۱. مقدار متغیرهای محلی اکتور

این اطلاعات زمانی در پیغام قرار داده می‌شوند که اکتور دیگری برای ادامه‌ی محاسبات نیاز به دسترسی به آنها داشته باشد. به عنوان مثال در رویکرد دوم محاسبه‌ی معدل (بخش ۲.۲.۳.۴) اکتور سابقه، با دریافت پیغام GPAInfoRequest مقدار نمره‌ی سابقه را در بخشی از پیغام قرار می‌دهد و برای اکتور درس ارسال می‌کند.

۲. نام (آدرس) اکتورهایی که با این اکتور در ارتباط هستند

اکتوری که پیغام برای آن ارسال می‌شود ممکن است برای ادامه‌ی پردازش، نیاز به دسترسی به اکتور دیگری داشته باشد. اگر فرستنده‌ی پیغام به این اکتور دسترسی داشته باشد، می‌تواند نام آن را در پیغام قرار دهد تا اکتور دیگر بتواند با آن ارتباط برقرار کند.

۳. قرار دادن مقصد نهایی درخواست در پیغام

اگر بخواهیم هر کدام از اکتورهای دریافت کننده‌ی پیغام بتوانند پاسخ نهایی درخواست را ارسال کنند، باید نام اکتور گیرنده‌ی پاسخ نهایی در قالب پیغام قرار داده شوند.

- منطق پردازش پاسخ(ها)

در صورتی که در مدل همکاری انتخاب شده، اکتور فرستنده‌ی پیغام‌های مربوط به درخواست، موظف به گرفتن پاسخ از آنها باشد، در این مرحله منطق پردازش پاسخ‌ها طراحی گردد. این منطق ممکن است شامل ارسال پیغام‌های جدید، یا بروزرسانی متغیرهای حالت باشد و در ساده‌ترین حالت ممکن است ارسال همان پاسخ به اکتوری دیگر باشد.

۵.۱.۵ طراحی سایر اکتورها

در صورتی که در طراحی منطق پردازش درخواست قبلی، اکتور مسئول پیغام پاسخ درخواست را بدون همکاری با سایر اکتورها ارسال کند، در این مرحله طراحی رخداد سیستمی مورد کاربرد انتخاب شده به اتمام می‌رسد و گام‌های طراحی برای رخداد سیستمی بعدی (یا مورد کاربرد بعدی) تکرار می‌شود. در غیر این صورت (در طراحی منطق پردازش درخواست، تصمیم به ارسال پیغام به سایر اکتورها گرفته شده باشد) طراحی اکتورهای گیرنده از گام "منطق پردازش درخواست" ادامه می‌یابد.

۲.۵ الگوهای طراحی

در این بخش تعدادی از الگوهایی که می‌توانند در طراحی به روش تبادل ناهمگام پیغام به کار گرفته شوند گردآوری و ارائه شده است. ذکر چند نکته‌ی مقدماتی در این بخش ضروری به نظر می‌رسد:

۱. در ابتدا باید تأکید شود که هر چند که نکات این بخش تحت عنوان الگوی طراحی ارائه شده‌اند، اما به معنای دقیق کلمه از دیدگاه مهندسی نرم‌افزار، همردیف الگوهای طراحی (مشابه الگوهای طراحی شیءگرا[۳۴]) نیستند. دلیل این امر این است که اولاً الگوهای طراحی باید دارای نامگذاری دقیق و نظام‌مند باشند تا بتوان از آنها به عنوان زبان مشترکی بین طراحان و برنامه‌نویسان روش مورد بحث استفاده کرد. ثانیاً الگوهای طراحی در اثر تجربیات طراحی و پیاده‌سازی‌های متعدد و ارزیابی‌های دقیق حاصل می‌شوند. این درحالی است که اولاً این بخش از این پژوهش، اولین تلاش در راستای استخراج الگوهای طراحی در این روش است که در آن بیشتر تجربیات در اثر طراحی و پیاده‌سازی یک سیستم آزمایشی به دست آمده است و به همین دلیل احتمالاً از شمول و پختگی کافی برخوردار نیست و ثانیاً نام‌گذاری این الگوها فرایند حساسی است که از حوزه‌ی این پژوهش خارج است.

۲. در بخش اول این فصل الگوهای طراحی منطق پردازش یک درخواست در اکتورها ارائه می‌شود. در اینجا منظور از درخواست، پیغامی است که آغاز کننده‌ی یک فعالیت در اکتور است. همان طور که در بخش ۱.۵ ذکر شد، برای یک اکتور در قبال دریافت یک درخواست دو رویکرد می‌توان متصور بود: حالت اول این است که اکتور مورد نظر به تنهایی بتواند عملیات مربوط به درخواست را اجرا کند، و حالت دوم این است که نیاز به همکاری با سایر اکتورها داشته باشد. طبیعتاً طراحی حالت اول بسیار ساده بوده و نیازی به ارائه‌ی الگو برای آن وجود ندارد. بنابراین در این فصل الگوهایی بررسی می‌شوند که در آنها بیش از یک اکتور در پردازش یک درخواست درگیر

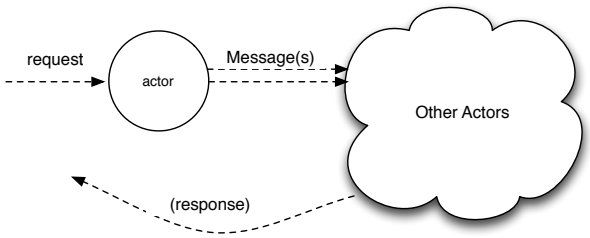
باشند.

۳. الگوهای ذکر شده ممکن است با الگوهای کلی طراحی همروند (مانند الگوی خط-لوله که در بخش ۱.۳ معرفی شد) در ارتباط باشند اما در این فصل بیشتر تأکید روی ارائه‌ی خواص الگوها و تناسب آنها با منطق دامنه است.

۴. تقسیم‌بندی الگوها با این دیدگاه مسئولیت اکتور دریافت کننده‌ی درخواست در پردازش منطق صورت گرفته است.

۱.۲.۵ دسته‌ی اول

دسته‌ی اول الگوهای طراحی، الگوهایی هستند که در آن اکتور دریافت کننده‌ی درخواست پس از ارسال پیغام‌های لازم مسئولیت دیگری در اجرای درخواست نخواهد داشت. شکل ۵.۵ حالت کلی این الگوها را نشان می‌دهد.



شکل ۵.۵: توصیف کلی الگوهایی که در آن اکتور دریافت کننده‌ی درخواست پس از ارسال پیغام(ها) مسئولیتی در پردازش درخواست ندارد.

۱.۱.۲.۵ الگوی ۱ (انتقال یا تحویل)

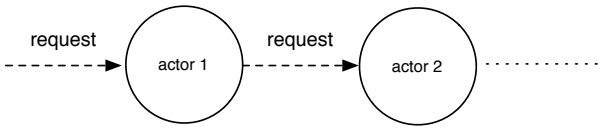
در این حالت درخواست مربوطه صرفاً پیغامی است که باید تحویل یکی از اکتورهایی شود که اکتور دریافت کننده‌ی درخواست به آن دسترسی دارد. این درخواست حالت اکتور دریافت کننده‌ی درخواست را عوض نمی‌کند ولی ممکن است حالت اکتور بعدی را عوض کند. شکل ۶.۵ این الگو را نمایش می‌دهد.

نحوه‌ی پیاده‌سازی:

اکتور با دریافت درخواست، همان پیغام درخواست شده را به اکتور مقصد ارسال می‌کند. تنها تغییری که ممکن است در پیغام رخ بدهد، نام پیغام است.

موارد استفاده:

این الگو در مواردی استفاده می‌شود که اکتور درخواست کننده به اکتور مقصد دسترسی مستقیم ندارد بلکه این دسترسی از طریق یک اکتور واسط فراهم می‌شود.



شکل ۶.۵: الگوی ۱ (انتقال یا تحویل)

۲.۱.۲.۵ الگوی ۲ (انتشار)

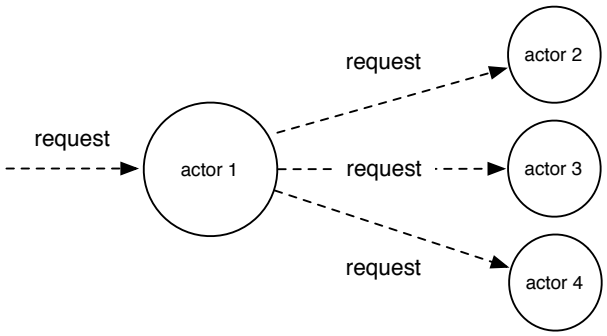
این الگو مشابه الگوی قبل است با این تفاوت که پیغام برای تعداد بیشتری از اکتورهای در ارتباط با اکتور جاری ارسال می‌شود. شکل ۷.۵ این الگو را نمایش می‌دهد.

نحوه‌ی پیاده‌سازی:

اکتور با دریافت پیغام، آن را برای اکتورهای مقصد به طور ناهمگام ارسال می‌کند.

موارد استفاده:

این الگو در مواردی استفاده می‌شود که اکتور دریافت کننده‌ی درخواست، لیستی از اکتورهای دیگر در اختیار دارد که اکتور درخواست کننده به طور مستقیم دسترسی به آنها ندارد.



شکل ۷.۵: الگوی ۲ (انتشار)

۳.۱.۲.۵ الگوی ۳ (وکالت)

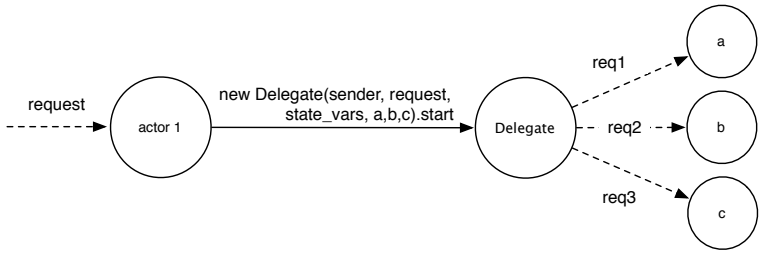
همان‌طور که در بخش ۱.۴.۱.۵ توضیح داده شد، در کاربردهای زیادی اکتور دریافت کننده‌ی درخواست تصمیم می‌گیرد کل عملیات مربوط به اجرای یک درخواست را به اکتور دیگری منتقل کند. الگوی ۳ این کاربرد را نشان می‌دهد. شکل ۸.۵ این الگو را نمایش می‌دهد.

نحوه‌ی پیاده‌سازی:

اکتور دریافت کننده‌ی درخواست، اکتور جدید را ایجاد می‌کند و اطلاعات لازم از جمله خود پیغام درخواست، فرستنده‌ی درخواست و متغیرهای حالت محلی و نیز دسترسی‌های لازم به سایر اکتورها را در اختیار اکتور جدید قرار می‌دهد. اینکه کدام یک از این اطلاعات برای انجام کار ضروری است بستگی به منطق دامنه دارد. در اکثر موارد عمر اکتور جدید محدود به پردازش درخواست است.

موارد استفاده:

این الگو زمانی استفاده می‌شود که اکتور اصلی دریافت کننده‌ی درخواست، به دلایلی از جمله کاهش پیچیدگی و یا حفظ قابلیت پردازش درخواست‌های همروند کل پردازش درخواست را به اکتور دیگری منتقل کند. در طراحی مورد کاربرد اخذ درس، پردازش پیغام درخواست اخذ درس مثالی از این الگو است (مراجعه کنید به بخش ۳.۳.۴).



شکل ۸.۵: الگوی ۳ (وکالت)

۴.۱.۲.۵ الگوی ۴

در این الگو اکتور برای پردازش درخواست، نیاز به ارسال پیغامی دیگر به یکی از اکتورهای در ارتباط با اکتور جاری است. تفاوت این نوع درخواست با درخواستی که در الگوی ۱ معرفی شد این است که در الگوی ۱ درخواست متعلق به اکتوری بود که اکتور دریافت کننده‌ی درخواست به آن دسترسی داشت. بر خلاف الگوی ۱، در این الگو درخواست به اکتور دریافت کننده مربوط است اما این اکتور تمام اطلاعات لازم برای پردازش پیغام را در اختیار ندارد (یا پردازش

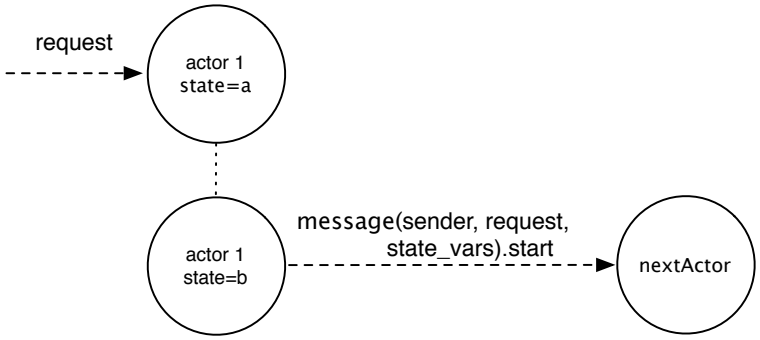
فقط مربوط به اکتور اول نیست). شکل ۹.۵ این الگو را نمایش می‌دهد.

نحوه‌ی پیاده‌سازی:

در این الگو، دریافت کننده‌ی پیغام، محاسبات مربوط به خود را انجام می‌دهد و اطلاعات لازم را در اختیار اکتور بعدی قرار می‌دهد.

موارد استفاده:

این الگو در مواردی که درخواست حالت مرحله‌ای دارد (مانند الگوی خط-لوله) مفید واقع می‌شود. مثالی از این الگو، درخواست بررسی قبولی درس، در مورد کاربرد اخذ درس (بخش ۱.۳.۳.۴) است.



شکل ۹.۵: الگوی ۴

۵.۱.۲.۵ الگوی ۵ (انشعاب)

در این الگو اکتور برای پردازش درخواست، نیاز به ارسال پیغام به اکتورهای دیگر است ولی تصمیم گرفته می‌شود که پاسخ‌های اکتورهای مذکور به اکتور دیگری فرستاده شود. در واقع به جای اینکه انشعاب و الحاق در یک اکتور صورت بگیرد، این دو مرحله از هم جدا شده‌اند. شکل ۱۰.۵ این الگو را نمایش می‌دهد.

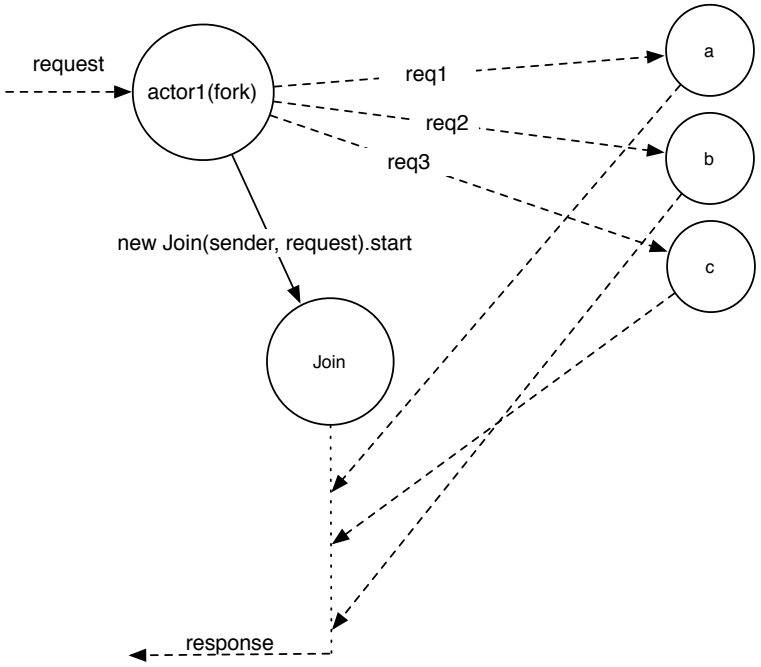
نحوه‌ی پیاده‌سازی:

عمل انشعاب به وسیله‌ی ارسال پیغام (ناهمگام) به تمام اکتورهای مربوطه انجام می‌پذیرد. در صورتی که نیاز به ایجاد اکتور الحاق وجود داشته باشد، اکتور دریافت کننده وظیفه دارد این کار را انجام دهد. اکتور دریافت کننده‌ی درخواست اطلاعاتی از قبیل آدرس اکتور الحاق و آدرس گیرنده‌ی نهایی پاسخ را به همراه پیغام‌ها به اکتورهای منشعب شده می‌فرستد.

موارد استفاده:

موارد استفاده‌ی این الگو به نوعی شبیه به الگوی ۳ (وکالت) است. یک انگیزه برای جدا کردن انشعاب و الحاق کم

کردن پیچیدگی اکتور مربوطه است. این عمل با تقسیم وظیفه‌ی انشعاب و الحاق بین دو اکتور مختلف صورت می‌پذیرد. انگیزه‌ی دوم امکان پردازش درخواست همروند برای اکتور انشعاب است. همان‌طور که توضیح داده شد، در این الگو مسئولیتی در قبال پاسخ اکتورهای منشعب شده ندارد. بنابراین می‌تواند با دریافت هر درخواست این عمل را تکرار کند و لزومی ندارد که منتظر پایان پردازش درخواست شود. در این الگو اکتوری که عمل الحاق را انجام می‌دهد اطلاعی از ترتیب ارسال درخواست‌ها ندارد، نمی‌تواند پاسخ‌ها را به ترتیب خاصی دریافت کند. به همین دلیل، موارد استفاده‌ی این الگو محدود به کاربردهایی است که ترتیب پیغام‌های یک درخواست اهمیتی نداشته باشد.



شکل ۱۰.۵: الگوی ۵

۲.۲.۵ دسته‌ی دوم

دسته‌ی دوم الگوهای طراحی، الگوهایی هستند که در آن مسئولیت اکتور دریافت کننده‌ی درخواست، با ارسال پیغام به سایر اکتورها پایان نمی‌پذیرد.

۱.۲.۲.۵ الگوی ۶ (انشعاب و الحاق بدون ترتیب)

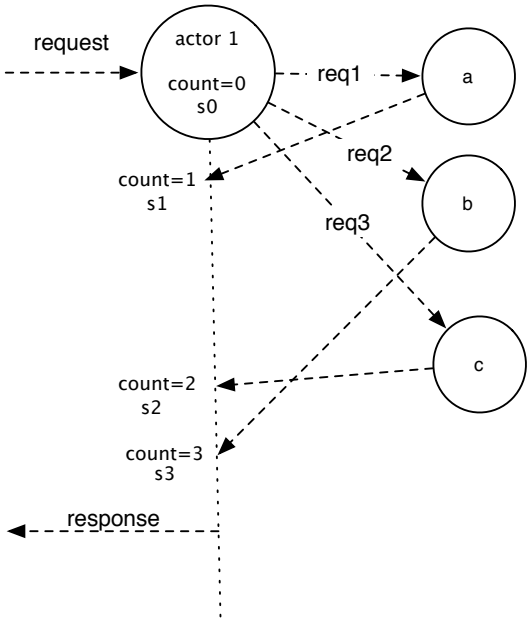
در این الگو اکتور برای پردازش درخواست، نیاز به ارسال پیغام به چند اکتور دیگر دارد و بعد از دریافت پاسخ از این اکتورها می‌تواند پاسخ درخواست را ارسال کند. فرض بر این است که ترتیب فرستادن پیغام‌ها و ترتیب دریافت پاسخ آنها اهمیت ندارد. شکل ۱۱.۵ این الگو را نمایش می‌دهد.

نحوه‌ی پیاده‌سازی:

اکتور با دریافت درخواست، پیغام‌هایی به همراه اطلاعات لازم برای اکتورهای مرتبط ارسال می‌کند. با توجه به اینکه ترتیب پاسخ‌ها اهمیتی ندارد، اکتور می‌تواند پاسخ‌ها را به هر ترتیبی پردازش کند. تعداد پاسخ‌های دریافت شده با یک متغیر محلی نگهداری می‌شود و با دریافت هر پاسخ، مقدار آن بروزرسانی می‌شود. علاوه بر این متغیر، سایر متغیرهای حالت سیستم نیز می‌توانند بروزرسانی شوند. با دریافت آخرین پیغام، منطق مربوط به پاسخ درخواست اجرا می‌شود و پاسخ ارسال می‌گردد.

موارد استفاده:

این الگو در حالاتی از منطق برنامه به کار می‌رود که درخواست مربوط به چندین اکتور است و یا پردازش آن منوط به کسب اطلاعات از چند اکتور است. علاوه بر آن، منطق دامنه باید این خاصیت را داشته باشد که ترتیب پاسخ‌های اکتورها مهم نباشد. مثالی از این الگو، طراحی اکتور محاسبه‌ی معدل در بخش ۲.۲.۳.۴ است.



شکل ۱۱.۵: الگوی ۶

۲.۲.۲.۵ الگوی ۷ (انشعاب و الحاق با ترتیب)

در این الگو، اکتور برای پردازش درخواست، نیاز به ارسال پیغام به چند اکتور دیگر دارد و بعد از دریافت پاسخ از این اکتورها می‌تواند پاسخ درخواست را ارسال کند. فرض بر این است که ترتیب فرستادن پیغام‌ها اهمیت ندارد ولی ترتیب دریافت پاسخ‌ها باید به ترتیب ارسال آنها باشد. شکل ۱۲.۵ این الگو را نمایش می‌دهد.

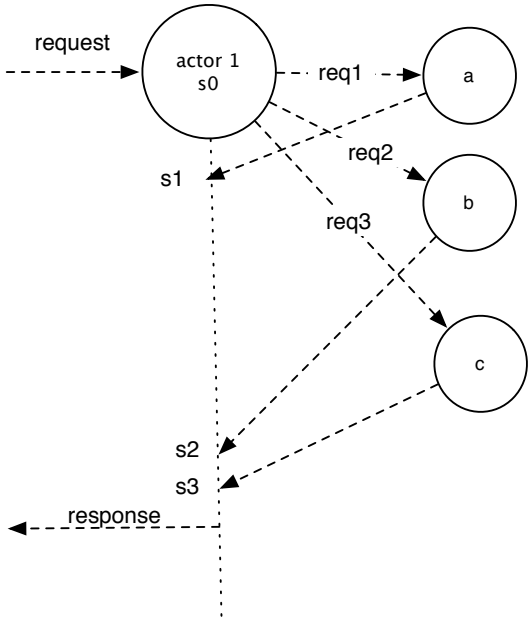
نحوه‌ی پیاده‌سازی:

با دریافت درخواست، پیغام‌هایی به همراه اطلاعات لازم برای اکتورهای مرتبط ارسال می‌کند. برای ایجاد قابلیت دریافت به ترتیب، می‌توان از قیود همگام‌سازی محلی (مراجعه کنید به ۲.۲.۳) استفاده کرد. به عنوان مثال، در کتابخانه‌ی اکتور اسکالا، این امکان وجود دارد که نتیجه‌ی ارسال هر پیغام را به صورت یک تابع ذخیره کنیم که با فراخوانی آن، اکتور برای دریافت پاسخ مربوطه متوقف شود[۷]

موارد استفاده:

این الگو در مواردی به کار گرفته می‌شود که ترتیب اجرای مراحل مهم باشند ولی اجرای هر مرحله به مرحله‌ی قبل وابسته نباشد. به عنوان مثال فرض کنید درخواست مورد نظر این باشد که به ترتیب برای چند نفر خرید سهام (انواع مختلف) انجام شود. قبل از انجام خرید هر نوع سهم لازم است قیمت آن در دست باشد. اکتور مورد نظر می‌تواند اطلاعات انواع سهام را به صورت ناهمگام از اکتورهای دیگر درخواست کند (انشعاب) اما با توجه به اینکه ترتیب خرید سهام مهم است، ترتیب پاسخ‌ها برایش اهمیت دارد. به همین دلیل در هر نوبت خرید، اطلاعات مربوط به سهام مربوطه را دریافت می‌کند.

[۷] به این امکان، *آینده* (Future) گفته می‌شود. (مراجعه کنید به بخش ۲.۲.۲.۲)



شکل ۱۲.۵: الگوی ۷

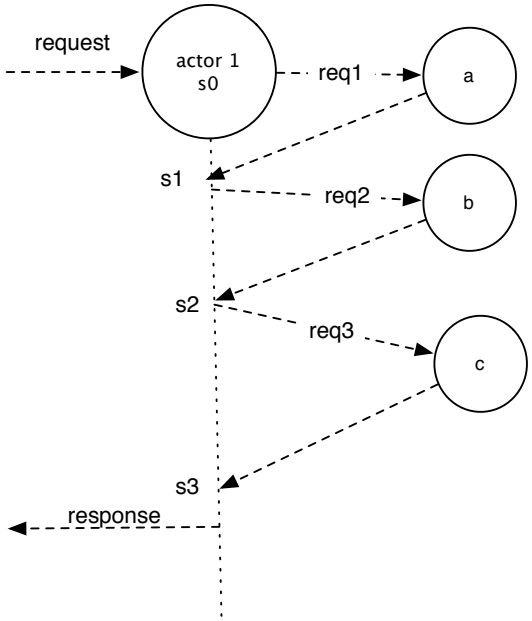
۳.۲.۲.۵ الگوی ۸ (فراخوانی مرتب)

در این الگو اکتور برای پردازش درخواست، نیاز به ارسال پیغام به چند اکتور دیگر دارد و بعد از دریافت پاسخ از این اکتورها می‌تواند پاسخ درخواست را ارسال کند. در این الگو فرض بر این است که قبل از دریافت پاسخ هر پیغام، پیغام بعدی را نمی‌توان ارسال کرد.

نحوه‌ی پیاده‌سازی:

با توجه به اینکه هر تبادل پیغام با اکتورهای دیگر باید به صورت مجزا انجام گیرد، مجبور به استفاده از تبادل همگام هستیم. بنابراین اکتور مورد نظر با ارسال هر پیغام، منتظر پیغام پاسخ می‌ماند و پس از اجرای منطق مربوط به آن می‌تواند پیغام بعدی را ارسال کند. **موارد استفاده:**

کاربرد این الگو در مواردی است که درخواست مربوط به یک کار مرحله‌ای باشد و هر مرحله از نظر منطق اجرا، به نتیجه‌ی مرحله‌ی قبل وابسته باشد. مانند اجرای مراحل یک الگوریتم که به صورت همروند نتوان آن را اجرا کرد. البته ممکن است هر پاسخی که اکتور منتظر آن می‌ماند صرفاً اعلام موفقیت یا شکست مرحله‌ی قبل باشد و در صورت شکست بخواهیم عملیات را متوقف کنیم (مانند فعالیت‌هایی که حالت تراکنشی دارند)



شکل ۱۳.۵: الگوی ۸

۳.۵ تجربیات و توصیه‌های طراحی و برنامه‌نویسی به روش تبادل ناهمگام پیغام

در انجام این پژوهش، رویکردهای مختلف طراحی به روش تبادل ناهمگام آزموده شده است. با توجه به نیاز به بررسی عملکرد صحیح طراحی و نیز ارزیابی کارایی و تغییرپذیری سیستم در این روش(فصل ۷)، پیاده‌سازی‌های متعددی نیز انجام شده است. در انجام این پیاده‌سازی‌ها نکات جالب توجه و مهمی مشاهده گردید که می‌توانند در پژوهش‌های مرتبط و نیز در استفاده از روش مطرح شده برای پیاده‌سازی‌های دیگر مورد استفاده قرار گیرند. در این بخش تعدادی از این تجربیات ارائه می‌گردد.

۱.۳.۵ طراحی قالب پیغام‌ها

در روش طراحی مبتنی بر تبادل ناهمگام پیغام، ارسال و دریافت تنها روش تبادل اطلاعات بین اکتورها است. اطلاعاتی که در قالب پیغام‌ها گذاشته می‌شود می‌توانند باعث ایجاد تغییرات عمده‌ای در روش طراحی بشوند. در این بخش چند نکته‌ی مفید برای طراحی قالب پیغام‌ها ذکر می‌شود:

اغلب وقتی یک اکتور درخواستی را برای اکتور دیگر ارسال می‌کند انتظار دریافت پاسخی نیز دارد. در زبان‌های

```
1 react {
2         case Message(x)
3         sender ! x+1
4 }
```

(الف)

```
1 react {
2         case Message(x, target)
3         target ! x+1
4 }
```

(ب)

شکل ۱۴.۵: پاسخ به درخواست از طریق الف) اشاره‌گر به فرستنده و ب)مقصد قرار داده شده در پیغام

مبتنی بر مدل اکتور، امکان پاسخ‌گویی به پیغام به صورت دستور به صورت دستور وجود دارد.[۸] علاوه بر این دستور، معمولا فرستنده‌ی یک پیغام از طریق کلمات کلیدی زبان قابل دسترسی هستند.[۹] به همین دلیل در حالت عادی نیازی به این حس نمی‌شود که وقتی پیغامی ارسال می‌شود، دسترسی به فرستنده در داخل پیغام قرار گیرد. اما **قرار دادن دسترسی به فرستنده در قالب پیغام** می‌تواند بسیار مفید باشد. این کار به این صورت انجام می‌شود که گیرنده‌ی پاسخ به جای آنکه همان فرستنده‌ی درخواست باشد، آرگومانی از پیغام دریافت شده باشد.

این رویکرد زمانی به کمک برنامه نویس می‌آید که در اثر تغییر طراحی، تصمیم گرفته می‌شود که گیرنده‌ی پاسخ پیغام عوض شود. در این صورت اگر پاسخ پیغام‌ها به اکتوری ارسال شده باشد که در قالب پیغام تغیین شده است، تغییر طراحی صرفاً در کلاس ارسال کننده‌ی درخواست صورت می‌گیرد (عوض کردن گیرنده‌ی پاسخ در قالب پیغام)، اما در صورتی که پاسخ پیغام‌ها به فرستنده‌ی پیغام صورت گرفته باشد این تغییر به تمام کلاس‌هایی که پیغام دریافت می‌کنند سرایت خواهد کرد.

۱. در حالتهایی که برای پاسخ به یک درخواست نیاز به برقراری ارتباط بین چند اکتور وجود دارد (مانند الگوی خط لوله) امکان دارد در طول مراحل پردازش پیغام‌ها بتوان پاسخ درخواست را قبل از اتمام کار تمام اکتورها ارسال کرد. به عنوان مثال فرض کنید یک مراحل پردازش یک درخواست شامل بررسی شروطی باشند که در صورت

[۸] مانند دستور reply در کتابخانه‌ی اکتور اسکالا

[۹] در کتابخانه‌ی اکتور اسکالا دسترسی به فرستنده‌ی پیغامی که در حال پردازش است از طریق کلمه‌ی کلیدی sender میسر می‌شود

برقرار نبودن شروط کل عملیات درخواست باید لغو شود. در چنین حالت‌هایی مطلوب است که اکتورهای میانی پایان بتوانند مستقیماً پاسخ درخواست را اجرا کنند و گیرنده را تا پایان پردازش درخواست منتظر نگذارند. برای عملی کردن این رویکرد باید اطلاعات گیرنده‌ی نهایی پاسخ درخواست در پیغام‌هایی که اکتورها ردوبدل می‌کنند موجود باشد. به همین دلیل **قرار دادن گیرنده‌ی نهایی پاسخ درخواست در پیغام‌های ردوبدل شده** می‌تواند مفید واقع شود.

۲. با اینکه پیغام در مدل تبادل پیغام به نوعی معادل متد در مدل طراحی شیءگرا است (از نظر تعریف واسط رفتاری شیء)، اما بین این دو تفاوت‌های زیادی وجود دارد

۲.۳.۵ خودداری از تفکر ترتیبی در طراحی

علیرغم شباهت‌ها و اشتراکات فراوان مدل شیءگرای ترتیبی با مدل تبادل ناهمگام پیغام، نحوه‌ی تفکر در طراحی این دو بسیار متفاوت است. به همین دلیل آغاز به طراحی در این روش برای کسی که سابقه‌ی طراحی شیءگرای ترتیبی را دارد قدری سخت است. از طرف دیگر بسیاری از پیاده‌سازی‌های مدل اکتور تمام مشخصه‌های معنایی اکتور را به طور کامل پیاده‌سازی نمی‌کنند[۸]. از جمله در کتابخانه‌ی اکتور اسکالا، این امکان وجود دارد که از یک اکتور به صورت یک شیء عادی استفاده کرد و متدهای آن را فراخوانی کرد. مجموع این عوامل باعث می‌شوند که طراح کم تجربه به طور ناخودآگاه به سمت طراحی ترتیبی سوق پیدا کند. طبیعتاً حل این مسئله نیازمند کسب تجربه و مهارت در طراحی همروند است. در این بخش نمونه‌هایی از مشکلاتی که در اثر تفاوت طراحی با استفاده از این دو دیدگاه به وجود می‌آیند معرفی می‌شود.

۱. همگام‌سازی بی‌مورد:

معمولاً در برخورد با مسائل طراحی اولین گزینه‌ای که به ذهن یک طراح کم تجربه می‌رسد استفاده از تبادل همگام پیغام است که شبیه به استفاده از فراخوانی متد در مدل شیءگرا است. این تفکر باعث می‌شود که در بسیاری از مراحل طراحی، بدون آنکه ذات مسئله نیازمند همگام سازی باشد از ارسال‌های همگام استفاده کنیم. برای جلوگیری از این مورد، در بررسی هر مسئله بهتر است طراح به این پرسش پاسخ دهد که آیا منطق مسئله نیازمند همگام‌سازی است یا خیر؟ در صورتی که منطق مسئله نیاز به همگام‌سازی نباشد، به احتمال زیاد لزومی به استفاده از تبادل همگام در برناه وجود ندارد.

۲. تبادل پیغام‌های بی‌مورد:

```scala
actorA !? Msg2(value) match {
    case Response2(r) =>
      // ...
}

receive {
    case Msg1(value) =>
      reply(Response1(value))
}
```

اکتور B

```scala
actorB !? Msg1(value) match {
    case Response1(r) =>
      // ...
}

receive {
    case Msg2(value) =>
      reply(Response2(value))
}
```

اکتور A

شکل ۱۵.۵: وقوع بن‌بست در تبادل پیغام بین دو اکتور

با اینکه پیغام در مدل تبادل پیغام به نوعی معادل متد در مدل طراحی شیءگرا است (از نظر تعریف واسط رفتاری شیء)، اما بین این دو تفاوت‌های زیادی وجود دارد. یکی از تفاوت‌های مهم این دو در این است که اگر متد خروجی‌ای را برگرداند، که در پیاده‌سازی یک منطق مورد استفاده قرار می‌گیرد، محل برگشت خروجی همان شیءای است که متد را فراخوانی کرده است. اما در تبادل پیغام، می‌توانیم خروجی را به اکتور دیگری ارسال کنیم. انتقال این تفکر ترتیبی به طراحی به روش تبادل پیغام می‌تواند باعث به وجود آمدن تبادل پیغام‌های بی‌مورد گردد. این مورد به طور کامل در رویکرد اول طراحی منطق مربوط به محاسبه‌ی معدل دیده می‌شود (بخش ۱.۲.۳.۴).

۳. تبادل پیغام منجر به بن‌بست اینکه مدل اکتور با توجه به عدم استفاده از به اشتراک گذاری حالت در حافظه، مشکلات همروندی کمتری نسبت به استفاده از ریسمان‌ها و حافظه‌ی مشترک دارد([۲۳])، ممکن است باعث این تفکر اشتباه بشود که در مدل اکتور نباید نگران مشکلات ناشی از کنترل همروندی باشیم. شکل ۱۵.۵ مثالی را نشان می‌دهد که دو اکتور در تبادل پیغام با یکدیگر دچار بن‌بست شده‌اند.

نکته‌ای که باید دقت کرد این است که در این مثال در صورتی که تبادل پیغام، به صورت ناهمگام انجام شود مشکل بن‌بست رخ نمی‌دهد. بنابراین اجتناب از تبادل همگام در کاهش حالات بن‌بست مفید می‌باشد. اما در تمام حالات امکان اجتناب از تبادل همگام وجود ندارد. به همین دلیل باید در هنگام طراحی و پیاده سازی به عدم

وجود چرخه‌های وابستگی بین درخواست‌های همگام دقت کرد.

علاوه بر مورد قبلی که مربوط به وجود چرخه در درخواست‌های همگام بود، یک نکته‌ی مهم دیگر در برنامه‌نویسی به روش تبادل پیغام وجود دارد. همان‌طور که در بخش قبل ذکر شد، تبادل پیغام در مدل اکتور، واسط رفتار اشیاء، معادل فراخوانی متد در مدل ترتیبی است. با همین تفکر ممکن است در قسمتی از طراحی یا پیاده‌سازی برنامه به روش تبادل پیغام، یک اکتور یک پیغام برای خودش ارسال کند. مثالی از این مورد در شبه‌کد شکل ۱۶.۵ نشان داده شده است. این مثال بخشی از کد یک اکتور در یک طراحی فرضی را نشان می‌دهد. این اکتور ۲ نوع پیغام دریافت می‌کند. اولین پیغام درخواست بررسی گذرانده شدن یک درس است که در جواب آن پاسخ مناسب داده می‌شود. منطق مربوط به طریقه‌ی فهمیدن اینکه درس گذرانده شده است یا خیر در شکل نشان داده نشده است. درخواست دوم بررسی گذرانده شدن پیش‌نیازهای یک درس است. با توجه به اینکه این اکتور به پیشنیازهای درس دسترسی دارد و قادر به پاسخگویی به درخواست بررسی گذرانده شدن درس نیز است، ممکن است برای بررسی گذراندن پیش‌نیازها این راهکار به ذهن برسد که اکتور به ازای هر درس پیش‌نیاز، یک درخواست بررسی گذرانده شدن درس برای خودش ارسال کند و پاسخ آنها را دریافت کند. این منطق در قالب متد hasPassedPreReqs نشان داده شده است. معادل این طراحی در مدل شیءگرای ترتیبی در شکل ۱۷.۵ نشان داده شده است.

با اینکه این دو شبه‌کد دقیقاً یک منطق را مدل می‌کنند، طراحی حالت تبادل پیغام در این وضعیت منجر به بن‌بست اکتور می‌شود. دلیل این امر این است که با توقف یک اکتور برای دریافت یک پیغام، ریسمان اجرای آن متوقف می‌شود و تا وقتی که پیغامی را که درخواست شده دریافت نکند، ادامه‌ی اجرای آن از سر گرفته نمی‌شود. در شکل ۱۶.۵ اکتور در خط ۱۳ تعدادی پیغام برای خودش ارسال می‌کند، در ادامه در خط ۱۷ با استفاده از دستور receive منتظر دریافت پاسخ می‌ماند. در این لحظه اجرای این اکتور تا دریافت پاسخ متوقف می‌شود. با وارد شدن هر پیغام در صندوق پیغام اکتور، پیغام جدید با پیغامی که در بلوک receive درخواست شده مقایسه می‌شود و اگر پیغام مورد نظر نبود منتظر دریافت پیغام‌های دیگر می‌شود. این در حالی‌است که دریافت پیغام‌هایی که در خط ۱۳ فرستاده شده، در بلوک دیگری از اکتور در خط ۲ دریافت می‌شود. با توجه به اینکه اجرای برنامه در بلوک خط ۱۷ متوقف شده، پس از آن نوبت اجرا هیچ‌گاه به بلوک خط ۱ نمی‌رسد و در نتیجه پیغام‌های مربوطه هیچگاه پاسخ‌داده نمی‌شوند. به این ترتیب اکتور در وضعیت بن‌بست قرار می‌گیرد.

بنابراین علاوه بر توجه به حالت‌های بن‌بست ایجاد شده در اثر وابستگی چرخشی اکتورها به همدیگر، ممکن است ارسال پیغام از یک اکتور به خودش نیز در حالت‌های خاص منجر به بن‌بست شود. البته این موارد به سادگی با عوض کردن رویکرد طراحی (در طراحی مورد کاربرد اخذ درس این مشکل وجود ندارد) و یا با دقت در استفاده از بلوک‌های دریافت

```
receive {
    case HasPassed(course)
        if (student has passed the course ...)
          sender ! Passed(true)
        else sender ! Passed(false)

    case HasPassedPreReqs(course)
       if(hasPassedPreReqs(course))
}

def hasPassedPreReqs(course: Course): Boolean = {
    for (pre <- course.preRequisites) {
      self ! HasPassed(pre, self)
    }

    for (pre <- course.preRequisites) {
      receive {
        case Passed(course, true) =>
        //...
        case Passed(course, false) =>
          return false
      }
    }
    return true;
  }
```

شکل ۱۶.۵: شبه‌کد اسکالا برای حالت وقوع بن‌بست به دلیل ارسال پیغام اکتور به خودش

```
1 public boolean hasPassedCourse(Course course) {
2   if student has passed this course
3     return true
4   else
5     return false
6 }
7
8 public boolean hasPassedPreReqs(Course course) {
9   fore(pre in course.preRequisites) {
10    if(! this.hasPassedCourse(pre)) {
11        return false;
12    }
13  return true
14  }
15 }
```

شکل ۱۷.۵: شبه‌کد جاوا برای حالت شیءگرای ترتیبی در شکل ۱۶.۵

پیغام قابل حل هستند.

فصل ۶

ارزیابی

۱.۶ مقدمه

در فصل‌های قبل روش طراحی منطق دامنه بر اساس تبادل ناهمگام پیغام ارائه شد و برخی از الگوهای طراحی در این روش به همراه نکات مهم آن مورد بررسی قرار گرفت. در این فصل دو معیار کیفی تغییرپذیری و کارایی در طراحی به این روش مورد ارزیابی قرار می‌گیرد. برای انجام ارزیابی ابتدا سیستم معرفی شده در بخش ۲.۴ بر اساس طراحی ارائه شده، پیاده سازی شده است. با توجه به اینکه قضاوت در مورد معیارهای انتخاب شده برای بررسی، در مقایسه با رویکرد طراحی شیءگرا مفهوم پیدا می‌کند سیستم مذکور توسط روش متداول شیءگرا نیز طراحی و پیاده‌سازی شده است. متن برنامه‌ی تولید شده به هر دو روش در بخش پیوست‌ها ارائه شده است. جهت حفظ دقت مقایسه تلاش شده است تا دو طراحی از نظر کارکرد حداکثر شباهت با یکدیگر را داشته باشند. برای پیاده‌سازی سیستم به روش طراحی بر اساس تبادل ناهمگام پیغام، از کتابخانه‌ی اکتور اسکالا استفاده شده و برای پیاده‌سازی به روش شیءگرا از زبان جاوا استفاده شده است.

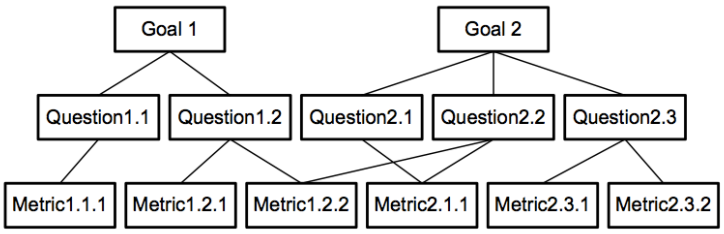
۲.۶ روش ارزیابی

در این ارزیابی از روش *هدف-پرسش-معیار*[۱] استفاده شده است. هدف-پرسش-معیار یک روش نظام‌مند برای تعیین معیارهای ارزیابی متناسب با دامنه‌ی مسئله است. این روش یک رویکرد بالا-به-پایین[۲] است که از قدم‌های زیر تشکیل می‌شود:[۳۵]

۱. تعیین هدف: به این پرسش پاسخ می‌دهد که **هدف** از اندازه‌گیری و ارزیابی چه چیزی است.

۲. قدم بعدی تعریف **پرسش**‌هایی است که اگر پاسخ آنها داده شود هدف اندازه‌گیری براورده می‌شود.

۳. برای پاسخ کمّی به پرسش‌هایی که تعریف شده‌اند، به هر پرسش دسته‌ای از **معیار**ها تخصیص داده می‌شود.



شکل ۱.۶: ساختار روش هدف-پرسش-معیار

۳.۶ ارزیابی تغییرپذیری

در این بخش به ارزیابی تغییرپذیری سیستم تحت طراحی بر اساس تبادل ناهمگام پیغام می‌پردازیم. برای ایجاد امکان ارزیابی کمّی تغییرپذیری، می‌توان از معیارهای کیفیت نرم‌افزار[۳] استفاده کرد. این معیارها خصوصیاتی از قبیل پیچیدگی، قابلیت استفاده‌ی مجدد و آزمون‌پذیری یک سیستم را به صورت کمّی قابل سنجش و مقایسه می‌کنند. در ادامه با توجه به روش هدف-پرسش-معیار، عناصر ارزیابی تغییرپذیری سیستم را تعیین می‌کنیم.

---

[۱] Goal-Question-Metric

[۲] top-down

[۳] Software Quality Metrics

۱.۳.۶ هدف

در این ارزیابی هدف ارزیابی تغییرپذیری طراحی به روش تبادل ناهمگام پیغام است. این ارزیابی از طریق مقایسه‌ی نسخه‌های پیاده سازی شده‌ی سیستم آموزش ساده (بخش و روش طراحی تبادل ناهمگام پیغام و طراحی متداول شیءگرا صورت می‌پذیرد.

۲.۳.۶ پرسش‌ها

برای این ارزیابی ۳ پرسش در نظر گرفته می‌شود:

۱. سیستم طراحی شده به روش تبادل ناهمگام، از نظر معیارهای کیفیت نرم‌افزار چگونه با سیستم طراحی شده به روش شیءگرا مقایسه می‌شود؟

۲. اعمال تغییر قواعد منطق برنامه[۴] روی کدام نوع طراحی مشکل‌تر است؟

۳. اعمال تغییر در مدل دامنه‌ی سیستم روی کدام نوع طراحی مشکل‌تر است؟

۳.۳.۶ معیارها

معیارهای مختلفی برای بررسی کیفیت برنامه‌های شیءگرا در پژوهش‌های مختلف ارائه شده‌اند که از آن جمله می‌توان به معیارهای ابریو[۵][۳۶، ۳۷]، معیارهای بانسیا و همکاران[۶][۳۸] و معیارهای چیدامبر و کمرر[۷] که به متریک‌های سی‌کی معروف هستند [۳۹]. در این پژوهش تعدادی از متریک‌های موجود برای ارزیابی انتخاب شده‌اند. این انتخاب بر اساس میزان ارجاع در پژوهش‌های مختلف به این متریک‌ها و نیز تناسب آنها با طراحی‌های این پژوهش انجام شده است. مقاله‌ی [۴۰] اطلاعات کاملی از متریک‌های مختلف کیفیت نرم‌افزار و میزان ارجاع به آنها را ارائه کرده است. معیارهای انتخاب شده برای پرسش اول ادامه معرفی شده‌اند. در تعاریف و توضیحات این معیارها از مقاله‌ی [۴۰] بهره گرفته شده است.

---

[۴] business rules

[۵] Abreu

[۶] Bansia et al

[۷] Chidamber and Kemerer

۱. **تعداد کلاس:** تعداد کلاس‌های پیاده‌سازی شده (بدون در نظر گرفتن کلاس‌های داخلی)

۲. **میانگین تعداد متد در هر کلاس**

۳. **میانگین تعداد فیلد تعریف شده در هر کلاس**

۴. **تعداد خطوط متن برنامه**

۵. **میانگین تعداد خطوط متن هر کلاس**

۶. **میانگین پیچیدگی چرخه‌ای (CC)**[۸]: پیچیدگی چرخه‌ای یک متد برابر است با تعداد نقاط تصمیم در گراف کنترل آن بعلاوه‌ی یک. نقاط تصمیم در یک متد در شرط‌ها و حلقه‌ها و دستورهای مشابه رخ می‌دهند. به عنوان مثال، پیچیدگی چرخه‌ای برای یک متد که یک بلوک شرط (if) یا یک حلقه دارد عدد ۲ است و برای متدی که صرفاً دستورات خطی دارد عدد ۱ است. هرچه عدد پیچیدگی چرخه‌ای بزرگتر باشد برنامه به موارد آزمون بیشتری نیاز دارد.

۷. **میانگین متدهای وزن‌دار در کلاس**[۹]: تعداد متدهای وزن‌دار در یک کلاس برابر است با جمع پیچیدگی چرخه‌ای متدهای آن کلاس.

۸. **میانگین عمق درخت وراثت (DIT)**[۱۰]: عمق درخت وراثت برابر است با تعداد کلاس‌هایی که در زنجیره‌ی وراثت یک کلاس آن را به ریشه می‌رساند.

۹. **میانگین تعداد فرزندان:** تعداد فرزندان یک کلاس برابر است با تعداد کلاس‌هایی که در زنجیره‌ی وراثت کلاس پایین‌تر از خود کلاس هستند.

۱۰. **میانگین جفت‌شدگی**[۱۱] **بین اشیاء:** یک کلاس با کلاس دیگر جفت‌شدگی دارد اگر یکی از دو کلاس (یا هردو) متدی از دیگری را فراخوانی کرده باشد. معیار جفت‌شدگی بین اشیاء برای یک کلاس برابر است با تعداد کلاس‌های که با کلاس مورد نظر جفت‌شدگی دارد. (معیار انتخاب شده میانگین این عدد در بین همه‌ی کلاس‌ها است.)

پرسش‌های دوم و سوم مربوط به اعمال تغییرات در قواعد منطق دامنه و مدل دامنه‌ی سیستم می‌باشند. برای این پرسش‌ها معیارهای زیر مناسب تشخیص داده شده‌اند:

---

[۸] Average Cyclomatic Complexity

[۹] Average Weighted Methds per Class

[۱۰] Average Depth of Inheritance Tree

[۱۱] Coupling

۱. تعداد خطوط اضافه شده به متن برنامه

۲. تعداد کلاس اضافه شده

۳. تعداد متد وزن‌دار اضافه شده

۴. تعداد کلاس تغییر داده شده

۵. تعداد متد تغییر داده شده

۴.۳.۶ نتایج ارزیابی

۱.۴.۳.۶ پرسش اول

برای پاسخ به پرسش اول، معیارهای انتخاب شده در پیاده‌سازی‌های سیستم آموزش ساده (بخش و روش تبادل ناهمگام پیغام و شیءگرا محاسبه شده و باهم مقایسه شده‌اند. نتیجه‌ی این مقایسه در جدول ۱.۶ ارائه شده است.

معیار	مقدار معیار برای طراحی به روش شیءگرا	مقدار معیار برای طراحی به روش تبادل ناهمگام پیغام
تعداد کلاس	۱۵	۱۷
میانگین تعداد متد در هر کلاس	۶/۲	۵/۱۱
میانگین تعداد فیلد تعریف شده در هر کلاس	۱/۹۳	۴/۲۹
تعداد خطوط متن برنامه	۶۶۴	۷۶۰
میانگین تعداد خطوط متن هر کلاس	۴۴/۲۷	۴۲/۲۲
میانگین پیچیدگی چرخه‌ای	۱/۵۸	۱/۲۵
میانگین متدهای وزن‌دار در کلاس	۹/۸	۷/۱۲
میانگین عمق درخت وراثت	۱/۳۳	۱/۵۹
میانگین تعداد فرزندان	۰/۳۳	۰/۵۹
میانگین جفت شدگی بین اشیاء	۳/۶	۳/۱۸

جدول ۱.۶: مقادیر معیارها برای پرسش اول

۲.۴.۳.۶ پرسش دوم

پرسش دوم مربوط به تغییر در قواعد منطق دامنه است. تغییر انتخاب شده به این نحو است که در مورد کاربرد اخذ درس (بخش ۲.۴) یک شرط به شروط لازم برای قبول اخذ درس اضافه می‌کنیم. این شرط مربوط به تعداد واحدهای اخذ شده توسط دانشجو در ترم جاری است. طبق این شرط، اگر تعداد واحدهای اخذ شده توسط دانشجو در ترم جاری با اخذ درس جدید به بیش از ۲۰ واحد برسد اجازه‌ی اخذ درس به دانشجو نمی‌دهیم.
نتایج اعمال تغییر ذکر شده در هر کدام از طراحی‌ها با توجه به معیارهای انتخاب شده، در قالب جدول ۲.۶ ارائه شده است.

معیار	مقدار معیار برای طراحی به روش شیءگرا	مقدار معیار برای طراحی به روش تبادل ناهمگام پیغام
تعداد خطوط اضافه شده به متن برنامه	۱۳	۱۹
تعداد کلاس اضافه شده	۰	۱
تعداد متد وزن‌دار اضافه شده	۴	۶
تعداد کلاس تغییر داده شده	۲	۲
تعداد متد تغییر داده شده	۱	۲

جدول ۲.۶: مقادیر معیارها برای پرسش دوم

۳.۴.۳.۶ پرسش سوم

پرسش سوم مربوط به تغییر در اشیاء مدل دامنه است. این پرسش با اعمال دو تغییر در سیستم پاسخ داده شده است:

۱. تغییر اول این است که یک کلاس با عنوان قواعد ترم[۱۲] به سیستم اضافه می‌شود. وظیفه‌ی این کلاس همان‌طور که از نام آن مشخص است تعیین قواعد یک ترم است. این قواعد شامل مواردی مثل سقف تعداد واحدهای اخذ شده، امکان اخذ مجدد درس و امکان اخذ درس بدون گذراندن پیش‌نیاز می‌شود. ارتباط این کلاس با بقیه‌ی کلاس‌های دامنه به این صورت است که هر شیء ترم، شیء قواعد ترم مربوط به خودش را دارد. فایده‌ی اضافه

[۱۲] TermRegulations

معیار	مقدار معیار برای طراحی به روش شیءگرا	مقدار معیار برای طراحی به روش تبادل ناهمگام پیغام
تعداد خطوط اضافه شده به متن برنامه	۴۳	۳۴
تعداد کلاس اضافه شده	۱	۱
تعداد متد وزن‌دار اضافه شده	۱۲	۶
تعداد کلاس تغییر داده شده	۲	۲
تعداد متد تغییر داده شده	۳	۳

جدول ۳.۶: مقادیر معیارها برای پرسش سوم (تغییر اول)

شدن این کلاس این است که امکان تغییر قوانین در ترم‌های مختلف در سیستم آموزشی به وجود می‌آید. نتایج اعمال تغییر ذکر شده در هر کدام از طراحی‌ها با توجه به معیارهای انتخاب شده، در قالب جدول ۳.۶ ارائه شده است.

۲. تغییر دوم به این صورت است که به مدل دامنه‌ی سیستم یک کلاس با عنوان برنامه[۱۳] اضافه می‌شود. این کلاس وظیفه دارد پیش‌نیازی بین دروس را تعیین کند. با این تغییر، رابطه‌ی پیش‌نیازی بین دروس حذف می‌شود و پیش‌نیاز بودن یک درس در قالب یک برنامه مشخص می‌شود. هر دانشجو برای تحصیل یک برنامه را انتخاب می‌کند و در آن برنامه مشخص می‌شود که پیش‌نیازهای یک درس کدام دروس هستند. نتایج اعمال تغییر ذکر شده در هر کدام از طراحی‌ها با توجه به معیارهای انتخاب شده، در قالب جدول ۴.۶ ارائه شده است.

[۱۳] Program

معیار	مقدار معیار برای طراحی به روش شیءگرا	مقدار معیار برای طراحی به روش تبادل ناهمگام پیغام
تعداد خطوط اضافه شده به متن برنامه	۲۸	۲۷
تعداد کلاس اضافه شده	۲	۲
تعداد متد وزن‌دار اضافه شده	۶	۶
تعداد کلاس تغییر داده شده	۲	۳
تعداد متد تغییر داده شده	۲	۳

جدول ۴.۶: مقادیر معیارها برای پرسش سوم (تغییر دوم)

۴.۶ ارزیابی کارایی

۱.۴.۶ بررسی معیارهای ایستا

با توجه به بزرگی سیستم مورد مطالعه، برای این مطالعه‌ی موردی دو مورد کاربرد از مجموعه‌ی مهم‌تری

۲.۴.۶ اعمال تغییرات

۱.۲.۴.۶ تغییر اول

۵.۶ نتایج ارزیابی

قبل از بررسی نتایج، لازم است برخی نکات در مورد اجرای آزمون‌ها مورد بررسی قرار گیرد. مطابق آن‌چه در فص

۱.۵.۶ تحلیل نتایج

با داشتن نتای

فصل ۷

جمع‌بندی و نکات پایانی

به عنوان جمع بندی متن حاضر، در این فصل به فهرستی از مهم‌ترین دستاوردهای این پژوهش خواهیم پرداخت. در مورد هر یک از این دستاوردها برخی نکات مهم نیز ذکر شده است. بعد از این، برخی از مهم‌ترین کاستی‌های چهارچوب ارائه شده آورده شده است. این کاستی‌ها در هر دو جنبه‌ی نظری و عملی مورد بررسی قرار گرفته‌اند. در نهایت، بر مبنای این موارد برخی جهت‌گیری‌های ممکن برای ادامه‌ی این پژوهش در آینده آورده شده است.

۱.۷ دستاوردهای این پژوهش

در این پژوهش روش طراحی منطق دامنه بر اساس تبادل ناهمگام پیغام مورد بررسی قرار گرفت. این روش طراحی با استفاده از مدل اکتور [۴] اشیاء سیستم را به فرایندهای فعالی که قادر به تبادل پیغام با یکدیگر هستند تبدیل می‌کند. بررسی صورت گرفته در این پژوهش به هدف استخراج نکات و الگوهای طراحی و مقایسه‌ی آن با رویکرد طراحی شیءگرا به صورت ترتیبی انجام گرفته است. در زیر برخی از مهمترین دستاوردهای این پژوهش آمده است:

- یک سیستم نمونه انتخاب شده و طراحی منطق دامنه‌ی آن به روش تبادل ناهمگام پیغام به طور کامل انجام شده است. ارائه‌ی روش طراحی به صورت مرحله‌ای و افزایشی باعث شده است تا بتوان از آن به صورت دستورالعملی برای طراحی همروند استفاده کرد.
- خروجی مهم پژوهش، روش‌ها و الگوهایی است که در این نوع طراحی کاربرد دارد. در هر الگوی استخراج شده،

روش پیاده‌سازی در مدل اکتور و کاربردهای الگو از نظر منطق دامنه بررسی شده است.

- تجربیاتی که در طراحی‌های صورت گرفته کسب شده به صورت قابل استفاده‌ای ارائه شده است و مطالعه‌ی این تجربیات، خواننده را با نکات ظریف و حساسی آشنا می‌کند که انجام طراحی به روش تبادل ناهمگام پیغام را بسیار ساده‌تر می‌کند.
- در ارزیابی روش طراحی ناهمگام، خصوصیات کیفی این روش از جمله تغییرپذیری و کارایی آن با روش طراحی شیءگرای ترتیبی مقایسه شده و نشان داده شده است که علاوه بر اینکه از نظر تغییرپذیری دو روش قابل مقایسه هستند، طراحی به روش تبادل ناهمگام پیغام در مواردی باعث افزایش چشم‌گیر کارایی سیستم می‌گردد.

۲.۷ جهت‌گیری‌های پژوهشی آینده

برخی از جهت‌گیریهای پژوهشی آینده برای تکمیل تحقیق حاضر در زیر آمده‌اند:

- در بررسی‌های صورت گرفته مشخص شد که برای ارزیابی کیفی طراحی شیءگرا به صورت ترتیبی معیارهای مختلفی وجود دارد که کیفیت برنامه را به صورت کمّی و قابل قیاس مشخص می‌کنند. با توجه به اینکه این معیارها بر اساس دیدگاه طراحی ترتیبی صورت گرفته و نکات و امکانات طراحی همروند در آنها نادیده گرفته شده است، نیاز به بازتعریف معیارهای موجود برای رویکرد طراحی بر اساس تبادل ناهمگام پیغام و نیز تعریف معیارهایی که مختص این رویکرد باشند کاملاً محسوس است. با توجه به نبود معیارهای کیفیت مختص سیستم‌های شیءگرای همروند، در این پژوهش برای انجام مقایسه‌ی کیفی معیارهای مشابه و قابل مقایسه با معیارهای طراحی ترتیبی استفاده شده است.
- مورد دیگری که در پژوهش‌های آینده می‌تواند مورد توجه قرار بگیرد تدوین الگوهای طراحی در روش تبادل ناهمگام پیغام است. در طراحی شیءگرا به روش ترتیبی این الگوها به صورت مدوّن موجود هستند[۳۴]. پژوهش حاضر با ارائه‌ی تعدادی از الگوهای موجود قدمی در انجام این مهم برداشته است اما مسلماً ارائه‌ی الگوهای طراحی در روش تبادل ناهمگام پیغام نیاز به بررسی پیاده‌سازی‌های متعدد در دامنه‌های مختلف دارد.

کتاب‌نامه

[1] J. pierre Briot, R. GUERRAOUI, K.-P. Löhr, and K. peter L, "Concurrency and distribution in object-oriented programming," tech. rep., 1998.

[2] C. Hewitt, *Description and Theoretical Analysis (Using PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot)*. Ph.D. thesis, Department of Computer Science, MIT, 1972.

[3] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott, "A foundation for actor computation," *J. Funct. Program.*, vol.7, no.1, pp.1–72, 1997.

[4] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass, 1986.

[5] G. Agha and C. Hewitt, "Concurrent programming using actors," pp.37–53, 1987.

[6] G. Agha, "Concurrent object-oriented programming," *Commun. ACM*, vol.33, no.9, pp.125–141, 1990.

[7] R. K. Karmani and G. Agha, "Actors," in *Encyclopedia of Parallel Computing*, pp.1–11, 2011.

[8] R. K. Karmani, A. Shali, and G. Agha, "Actor frameworks for the jvm platform: a comparative analysis," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, (New York, NY, USA), pp.11–20, ACM, 2009.

[9] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha, "Evaluating ordering heuristics for dynamic partial-order reduction techniques," in *FASE*, pp.308–322, 2010.

[10] W. Kim and G. Agha, "Efficient support of location transparency in concurrent object-oriented programming languages," in *SC*, 1995.

[11] P.-H. Chang and G. Agha, "Towards context-aware web applications," in *DAIS*, pp.239–252, 2007.

[12] V. A. Korthikanti and G. Agha, "Towards optimizing energy costs of algorithms for shared memory architectures," in *SPAA*, pp.157–165, 2010.

[13] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, second ed. , 1996.

[14] P. Haller and M. Odersky, "Actors that unify threads and events," in *Coordination Models and Languages*, vol.4467 of *Lecture Notes in Computer Science*, pp.171–190, Springer Berlin / Heidelberg, 2007.

[15] E. A. Lee, "Overview of the ptolemy project," Tech. Rep. UCB/ERL M03/25, University of California, Berkeley, 2003.

[16] C. A. Varela and G. Agha, "Programming dynamically reconfigurable open systems with salsa," *SIGPLAN Notices*, vol.36, no.12, pp.20–34, 2001.

[17] L. V. Kale and S. Krishnan, "Charm++: a portable concurrent object oriented system based on c++," *SIGPLAN Not.*, vol.28, pp.91–108, Oct. 1993.

[18] M. Astley, "The actor foundry: A java-based actor programming environment," Open Systems Laboratory, Uni- versity of Illinois at Urbana-Champaign, 1998-99.

[19] Microsoft Corporation, "Asynchronous agents library," http://msdn.microsoft.com/en-us/library/dd492627.aspx.

[20] B. V. Martin Odersky, Lex Spoon. *Programming In Scala*. WALNUT CREEK, CALIFORNIA: artima, 2 ed. , 2010.

[21] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1996.

[22] M. Welsh, D. Culler, and E. Brewer, "Seda: an architecture for well-conditioned, scalable internet services," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, (New York, NY, USA), pp.230–243, ACM, 2001.

[23] John Ousterhout, "Why threads are a bad idea (for most purposes)," Invited talk at USENIX, January 1996.

[24] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *IN HOTOS*, 2003.

[25] B. Chin and T. Millstein, "T.d.: Responders: Language support for interactive applications," in *In: Proc. ECOOP*, pp.255–278, 2006.

[26] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol.410, no.2â€“3, pp.202 – 220, 2009. <ce:title>Distributed Computing Techniques</ce:title>.

[27] T. H. Feng and E. A. Lee, "Scalable models using model transformation," 2008.

[28] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman, "Abstraction and modularity mechanisms for concurrent computing," *IEEE Parallel and Distributed Technology: Systems and Applications*, vol.1, pp.3–21, 1993.

[29] T. Papaioannou, "On the structuring of distributed systems : the argument for mobility.," 2000.

[30] S. Frølund. *Coordinating distributed objects: an actor-based approach to synchronization*. Cambridge, MA, USA: MIT Press, 1996.

[31] A. Kay, "Prototypes vs classes," email listing `http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html`, October 1998.

[32] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[33] A. M. Wirfs-Brock. *Object Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley, 2003.

[34] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 ed. , November 1994.

[35] V. Basili, G. Caldeira, H. D. Rombach, *Encyclopedia of Softwar Engineering*, chap. The Goal Question Metric Approach. Wiley, 1994.

[36] F. B. Abreu and R. Carapua, "Candidate metrics for oos within taxonomy framework," in *Journal of System Softwrae*, vol.26, 1994.

[37] F. B. Abreu, "The mood metrics set," in *ECOOP'95, Workshop on Metrics*, 1995.

[38] J. Bansiya and C. Davis, "A hierarchical model for object-oriented design quality assessment," *Software Engineering, IEEE Transactions on*, vol.28, pp.4 –17, jan 2002.

[39] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol.20, pp.476 –493, jun 1994.

[40] A. Shaik, D. C. R. K. Reddy, and D. A. Damodaram, "Article: Object oriented software metrics and quality assessment: Current state of the art," *International Journal of Computer Applications*, vol.37, pp.6–15, January 2012. Published by Foundation of Computer Science, New York, USA.

واژه‌نامه‌ی فارسی به انگلیسی

ارلانگ ..... erlang
استثناء ..... exception
استدلال ..... reason
افزایشی ..... incremental
اکتور ..... actor
انصاف ..... fairness
ایستا ..... static
آینده ..... Future
بررسی گونه‌ها ..... type checking
بن‌باز ..... livelock
بی‌قاعده ..... Irregular
پراکنده ..... sparse
پشته ..... stack

تابعی ..... functional
تجزیه ..... decomposition
تجزیه‌ناپذیر ..... atomic
ترتیبی ..... sequential
تعویض متن ..... context switch
تقسیم-و-حل ..... devide and conquer
جفت‌شدگی ..... coupling
حالت مشترک ..... shared state
خط لوله ..... pipeline
رفتار ..... behavior
رویداد-بنیان ..... event-based
ریسمان ..... thread
ریسمان-بنیان ..... thread-based
زمان‌بندی ..... scheduling
شیء ..... object
شیء-بنیان ..... object-based
شیءگونه ..... object-style
غیرقطعی ..... non-deterministic, indeterminate
لفافه‌بندی‌شده ..... encapsulated
مجری رویداد ..... event handler
مسدود کننده ..... blocking
معناشناسی ..... semantics
مقیاس‌پذیر ..... scalable
مورد کاربرد ..... use case
وارونگی کنترل ..... inversion of control
همروند ..... concurrent

Design of Domain Logic Using Asynchronous Message Passing

Abstract

In recent years, interest in Actor model has been growing, among researchers as well as practitioners. This interest is triggered by emerging programming platforms such as multicore computers and cloud computers. In some cases, such as cloud computing, the Actor model is a natural programming model because of the distributed nature of these platforms. This trend in using concurrent programming using actors, makes the need for providing design principles and patterns in this model just like they are provided thoroughly in sequential object-oriented design books. In this research, we choose a simple domain model named simple educational system and take the design steps needed to implement it using asynchronous message passing. The extracted patterns of actor interactions and messaging styles are provided to be used in simillar design attempts. Moreover, an empirical evaluation of software quality metrics for the design is undertaken and the results are compared with a sequential oop approach for the same domain model.

Keywords: *asynchronous message passing, design patterns, object-oriented design, domain modeling*

University of Tehran

School of Electrical and Compuer Engineering

Design of Domain Logic Using Asynchronous Message Passing

by

Vahid Zoghi

Under supervision of

Dr. Ramtin Khosravi

A thesis submitted to the Graduate Studies Office
in partial fulfillment of the requirements
for the degree of M.Sc

in

Computer Engineering

Sep 2012