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 آنان که در خوشی‌هایم همراهی کردند و در ناخوشی‌هایم صبر؛

پدرم، مادرم و همسر مهربانم

قدردانی

خدای سبحان را سپاس می‌گویم که به من توان و قوه‌ی ذهنی عطا فرمود تا از عهده‌ی مشکلات انجام این پژوهش برآیم.

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

آزمون مبتنی بر مدل، به معنی تولید خودکار موارد آزمون از مدل کارکردی و صوری سیستم تحت آزمون به صورت جعبه سیاه، که به عنوان راه‌حلی برای مسئله‌ی تولید و اجرای خودکار آزمون‌ها مطرح شده است، استفاده‌ی روزافزونی درسال‌های اخیر داشته است. ایده‌ی آزمون مبتنی بر مدل، اساساً به هدف آزمون سیستم‌های با رفتار پیچیده (و معمولاً همروند) مطرح شده است. با این حال، در این روش‌ها مدل‌سازی رفتار سیستم توسط نمادگذاری‌های سطح پایین (مانند ماشین‌های گذار) انجام می‌شود. برای مثال، در این روش‌ها مدل‌سازی مقادیر داده‌ای به عنوان پارامترهای ورودی و خروجی سیستم تحت آزمون یا اصلاً امکان‌پذیر نیست و یا منجر به تولید مدل‌های بسیار پیچیده‌ای می‌شود.

از سوی دیگر، روش‌هایی برای آزمون خودکار نرم‌افزار معرفی شده‌اند که هدف آن‌ها تولید موارد آزمون شامل داده است. این روش‌ها به جای توصیف رفتار سیستم توسط ماشین‌های گذار، با استفاده از متن برنامه (به صورت جعبه سفید) امکان توصیف روابط میان مقادیر داده‌ای و چگونگی تولید مقادیر داده‌ای به هدف آزمون را مهیا می‌کند.

بر این اساس، در این پژوهش چهارچوبی یک‌پارچه برای مدل‌سازی همزمان رفتارهای مورد انتظار از سیستم تحت آزمون از یک سو و توصیف داده‌های ورودی و خروجی آن از سوی دیگر ارائه شده است. این چهارچوب برای توصیف این موارد از زبان یوام‌ال استفاده می‌کند. به این ترتیب امکان توصیف سیستم‌هایی که هم از نظر رفتاری و هم از نظر داده‌هایی که مبادله می‌کنند، پیچیده هستند فراهم می‌شود. در راستای طراحی این چهارچوب، هم‌چنین آزمون‌گری پیاده‌سازی شده است که موارد آزمون خود را بر اساس مدل‌های یوام‌ال به صورت خودکار تولید می‌کند. **واژه‌های کلیدی:** *طراحی منطق دامنه، تبادل ناهمگام پیغام، مدل بازیگر، همروندی*

فهرست مطالب

فهرست تصاویر

فصل ۱

مقدمه

از دیدگاه مهندسی نیازمندی ن

۱.۱ انگیزه‌ی پژوهش

VIII. CURRENT STATUS AND PERSPECTIVE Actor languages have been used for parallel and dis- tributed computing in the real world for some time (e.g. Charm++ for scientific applications on supercomputers, Er- lang for distributed applications). In recent years, interest in actor-based languages has been growing, among researchers as well as practitioners. This interest is triggered by emerg- ing programming platforms such as multicore computers, cloud computers, Web services, and sensor networks. In some cases, such as cloud computing, web services and sensor networks, the Actor model is a natural programming model because of the distributed nature of these platforms. As multicore architectures are scaled, multicore computers will also look more more like the traditional multicomputer platforms. This is illustrated by the prototype, 48-core Single-Chip Cloud Computer (SCC) developed by Intel [36]. However, the argument for using actor-based programming languages is not simply that they provide a good match for representing computation on

a variety of parallel and dis- tributed computing platforms. The point is that by extending object-based modeling to concurrent agents, actors provide a good starting point for simplifying the task of parallel (distributed, mobile) programming.

(از مقاله‌ی آقا ۲۰۱۰) روش‌های آزمون مبتنی بر مدل کاستی‌هایی نیز دارند. برای مثال در آی‌اوکو، استفاده از ماشین گذار برای توصیف سیستم تحت آزمون می‌تواند منجر به تولید مدل‌های بسیار پیچیده‌ای شود، زیرا ماشین‌های گذار علی‌رغم قدرت بیان بالا، چنان‌چه خواهیم دید، از نظر توصیفی نمادگذاری سطح پایینی محسوب می‌شوند و بنابراین مدل‌سازی جزئیات سیستم ممکن است حجم زیادی از پیچیدگی را در مدل‌ها به وجود آورد.

۲.۱ صورت مسئله

تمرکز اصلی این پژوهش بر آزمون مبتنی بر مدل سیستم‌های وابسته به داده[۱] قرار دارد. سیستم‌های وابسته به داده معمولاً حجم زیادی از اطلاعات را با محیط خود مبادله می‌کنند و رفتار آن‌ها وابسته به محاسباتی است که بر روی مقادیر داده‌ای انجام می‌دهند.

۳.۱ روش پژوهش

ارزیابی عملی با مطالعه موردی

۴.۱ روش ارزیابی

GQM

[۱] data dependent

۵.۱ خلاصه‌ی دستاوردهای پژوهش

برخی از دستاوردهای این پژوهش را می‌توان به این ترتیب برشمرد: داد. به این منظور نمای سطح بالا برای تولید آزمون‌ها و بررسی نتایج در روش اصلی آزمون مبتنی برمدل در شکل

۶.۱ ساختار پایان‌نامه

برای بررسی این موارد، ساختار این متن در ۶ فصل تنظیم گردیده است:

- به طور خلاصه مورد بررسی قرار گرفته‌اند.
- بررسی
- ه‌اند.
-

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.

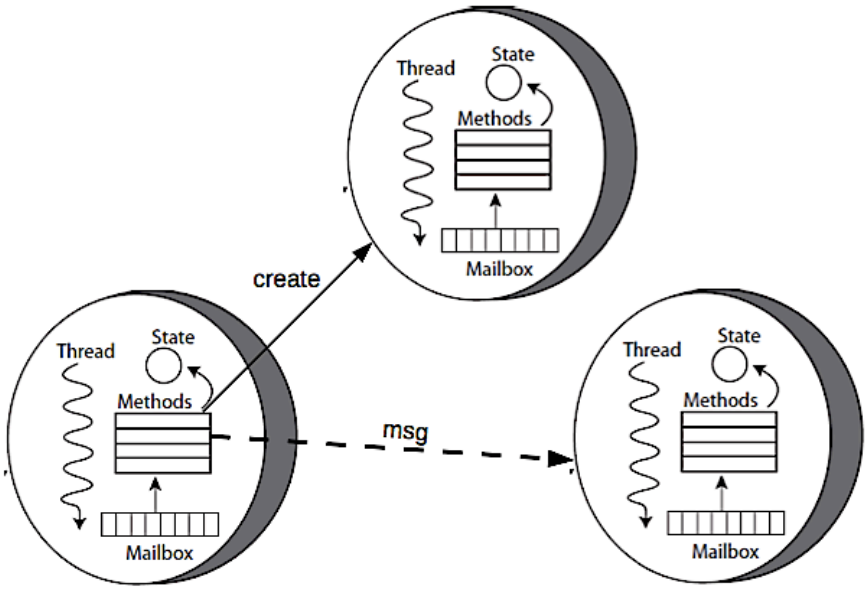
۱.۲ مدل اکتور

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. لفظ 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 بازیگر 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 (چرا؟) که در بخش ؟ معرفی 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

```
class Course(var id: String, var name: String, var units: Int,
  var preRequisites: List[Course]) extends BaseDomain {

  override def equals(other: Any): Boolean =
    other match {
      case that: Course =>
          id == that.id
      case _ => false
    }

  def printPrerequisites() = {
    for (pre <- preRequisites)
      println(pre)
  }

  override def toString = "[id= " + id + ",name=" + name + ",units=" + units + "]"
}
```

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 .. match .. case 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 toString، override 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

```
1 import scala.actors._
2 object SillyActor extends Actor {
3   def act() {
4     for (i <- 1 to 5) {
5       println("I'm acting!")
6       Thread.sleep(1000)
7     }
8   }
9 }
```

شکل ۳.۲: کد یک اکتور ساده در زبان اسکالا

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.

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 آرایه‌ای است که هر عضو آن یک تابع است. با فراخوانی هر تابع، اکتور تا دریافت پاسخ متناظر متوقف می‌شود. برای برداشتن پیغام از صندوق پیغام‌ها، از دو دستور receive و react استفاده می‌شود (تفاوت این دو دستور در بخش 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

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 بازیگر

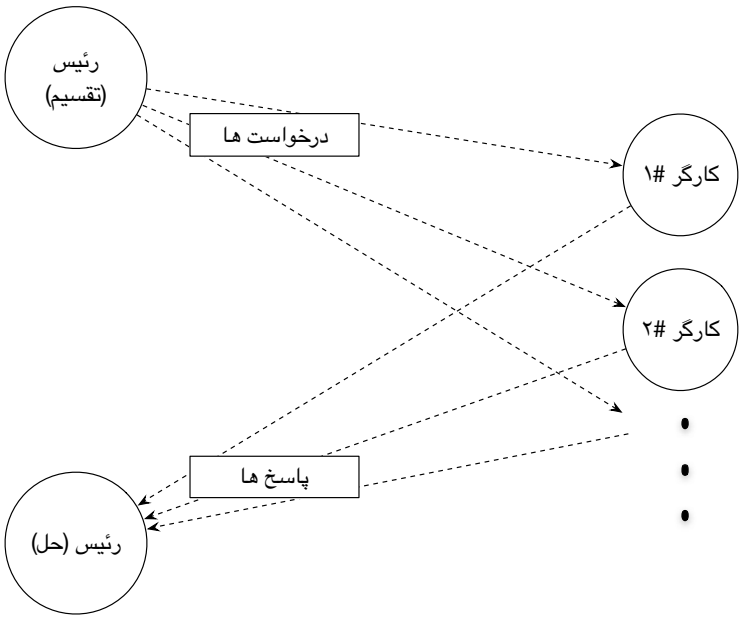
در برنامه‌نویسی همروند با بازیگرها دو نوع الگوی کلی معرفی شده است [۶]: یکی *تقسیم-و-حل*[۱] و دیگری *خط لوله*[۲]. در روش تقسیم-و-حل مسئله‌ی مورد بحث به زیربخش‌های کوچکتر و مستقل تقسیم می‌شود که هرکدام به صورت مستقل حل می‌شوند و نتایج هر زیربخش برای نتیجه‌گیری کلی ادغام می‌شوند. در برنامه‌نویسی به مدل بازیگر، برای پیاده‌سازی این الگو یک بازیگر رئیس[۳] در نظر گرفته می‌شود که تعدادی بازیگر 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

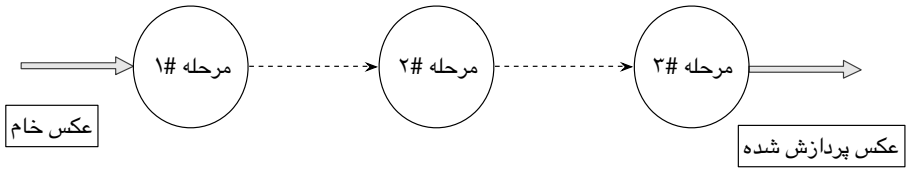


شکل ۱.۳: شمای کلی از الگوی تقسیم-و-حل در مدل بازیگر

کارگرها به نوبه‌ی خود منطق لازم برای حل زیر بخش را ایجاد نموده و نتیجه را به صورت پیغام دیگری برای بازیگر رئیس ارسال می‌کنند. نهایتا رئیس با ادغام نتایج جواب نهایی مسئله را تولید می‌کند. شایان ذکر است که فازهای تقسیم و حل لزوما توسط بازیگر یکسان اجرا نمی‌شوند. ممکن است اجرای فاز حل به بازیگر دیگری سپرده شود.[۲۷] مثال دیگری از پیاده‌سازی الگوی تقسیم-و-حل در مدل بازیگر 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 بازیگر را نمایش می‌دهد.
الگوی خط لوله برای حالت‌هایی مناسب است که فعالیت قابل تقسیم به بخش‌های افزایشی باشد. در این صورت هر بازیگر تغییرات مربوطه را در مدل ایجاد می‌کند و آن را به عنوان پیغام به بازیگر 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 بازیگر، هر مرحله به صورت یک بازیگر 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 بازیگرها

همان‌طور که در بخش‌های قبل ذکر شد، مدل بازیگر دارای خاصیت ناهمگامی است و ترتیب پیغام‌هایی که یک بازیگر دریافت می‌کند وابسته به ترتیب فرستاده شدن پیغام‌ها نیست. نتیجه‌ی این خاصیت این است که تعداد ترتیب[۶]های دریافت پیغام‌ها در مدل بازیگر نمایی است[۷]. به دلیل اینکه فرستنده‌ی پیغام از حالت محلی بازیگر 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 بازیگرهایی فرستاده شده و نتایج آن دریافت می‌شود ولی در آن ترتیب دریافت نتایج اهمیت داشته باشد. نیاز به این نوع اولویت‌بندی‌ها در مدل بازیگر 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 بازیگر، بازیگرها برای همگام‌سازی باهم ارتباط برقرار می‌کنند. در این قسمت دو نوع الگوی هماهنگی بازیگرها 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 بازیگر در این مدل به ترتیب زیر است:

۱. بازیگر فرستنده درخواست را در قالب یک پیغام به بازیگر 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 بازیگر 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 بازیگر را نشان می‌دهد. ارتباط شبه-آرسی‌پی در دو نوع سناریوی خاص مفید و ضروری است: یک سناریو این است که بازیگر نیاز به ارسال پیغام به صورت ترتیبی به یک یا چند بازیگر خاص دارد و تا حاصل شدن اطمینان از رسیدن پیغام قبلی پیغام بعد را ارسال نمی‌کند. سناریوی دوم این است که حالت[۹] بازیگر فرستنده بستگی به محتوای پاسخ دارد. در این حالت بازیگر 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 بازیگر 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 بازیگر به صورت امکانات سطح زبان وجود دارد[۸].

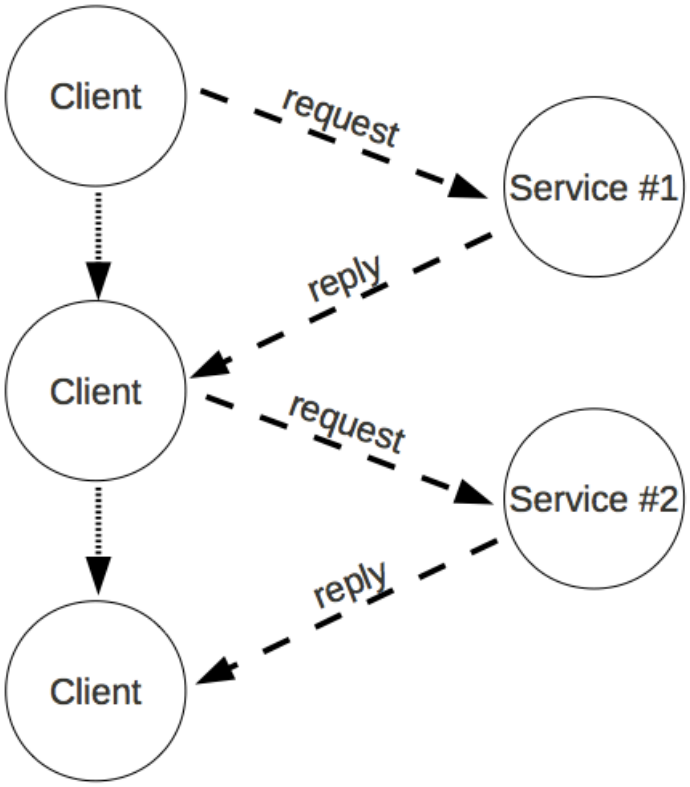
---

[۹] state

[۱۰] procedure

[۱۱] sequential

[۱۲] live lock



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

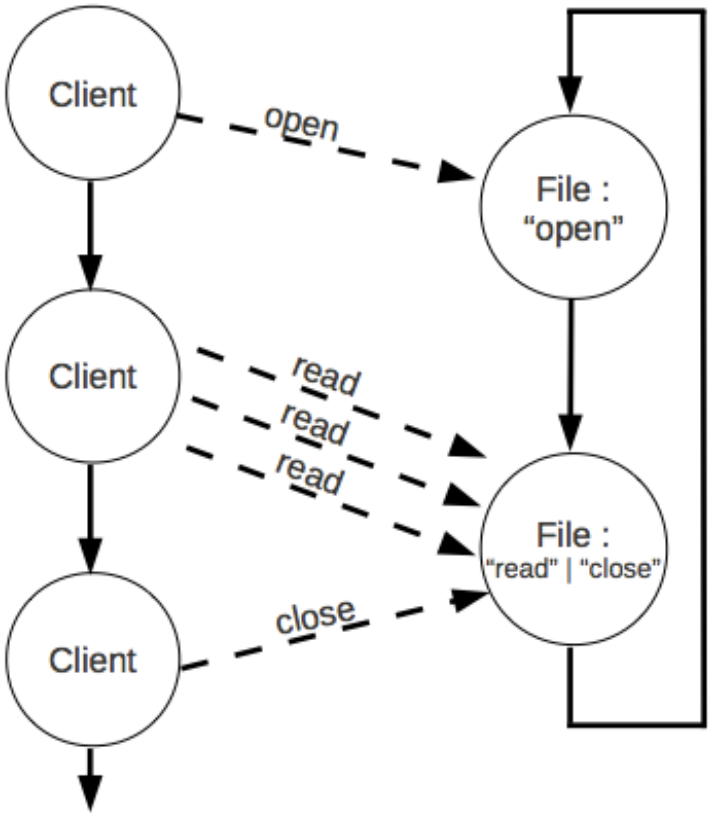
۲.۲.۳ قیود همگام‌سازی محلی

استفاده از قیود همگام‌سازی محلی روشی برای اولیت‌بندی پردازش پیغام‌ها در مدل بازیگر است[۳۰]. برای توضیح مفهوم همگام‌سازی محلی مثالی در شکل ۴.۳ ارائه شده است. در این مثال بازیگر 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 بازیگر امکانات مناسبی برای پشتیبانی از قیود همگام‌سازی محلی ارائه داده‌اند. به عنوان مثال در کتابخانه‌ی بازیگر 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. بازیگر 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 به روش 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 موجودیت‌های اصلی

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

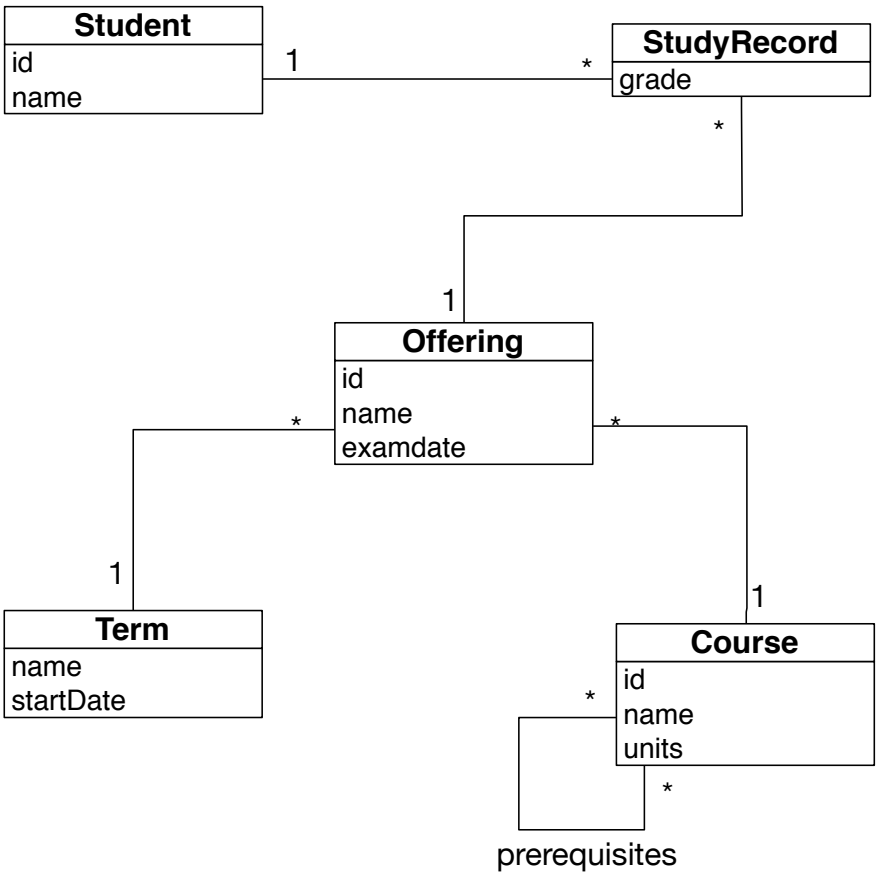
[۷] Term

[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 اصلی

منظور از اکتوران اصلی سیستم همان موجودیت‌های اصلی‌ای هستند که در بخش ۲.۲.۴ معرفی شدند. دلیل استفاده از واژه‌ی *اصلی* این است که احتمالاً علاوه بر این اکتورها، اکتوران دیگری نیز برای پیاده‌سازی کارکردهای سیستم لازم خواهد شد. در طراحی اکتوران اصلی صرفا فیلدهای 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

---

[۱۲] incremental

[۱۳] receipe

[۱۴] association

```
class Student(
  var id: String,
  var name: String,
  var studyRecords: List[StudyRecord]) extends Actor {

  override def act() {
    loop {
      react {
       case HasPassed(course, target) =>
          ...
       case HasTaken(course, target) =>
          ...
       case GPARequest(term: Term, target: Actor) =>
          ...
       case TakeCourse(offering, target) =>
      }
    }
  }
}
```

شکل ۲.۴: ساختار کلاس اکتور دانشجو

پیغام به آنها یک فیلد از نوع لیست سابقه نیز در کلاس دانشجو وجود دارد. قطعه کد ۲.۴ طرح ابتدایی کلاس 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.

```
class StudyRecord(
  var grade: Double,
  var offering: Offering) extends Actor {
  def act() {
    loop {
      react {
        case ...
      }
    }
  }
```

شکل ۳.۴: ساختار کلاس اکتور سابقه

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

```
class Term(
  var name: String,
  var startDate: Date) extends Actor {
  def act() {
    loop {
      react {
        case ...
      }
    }
  }
```

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.

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:

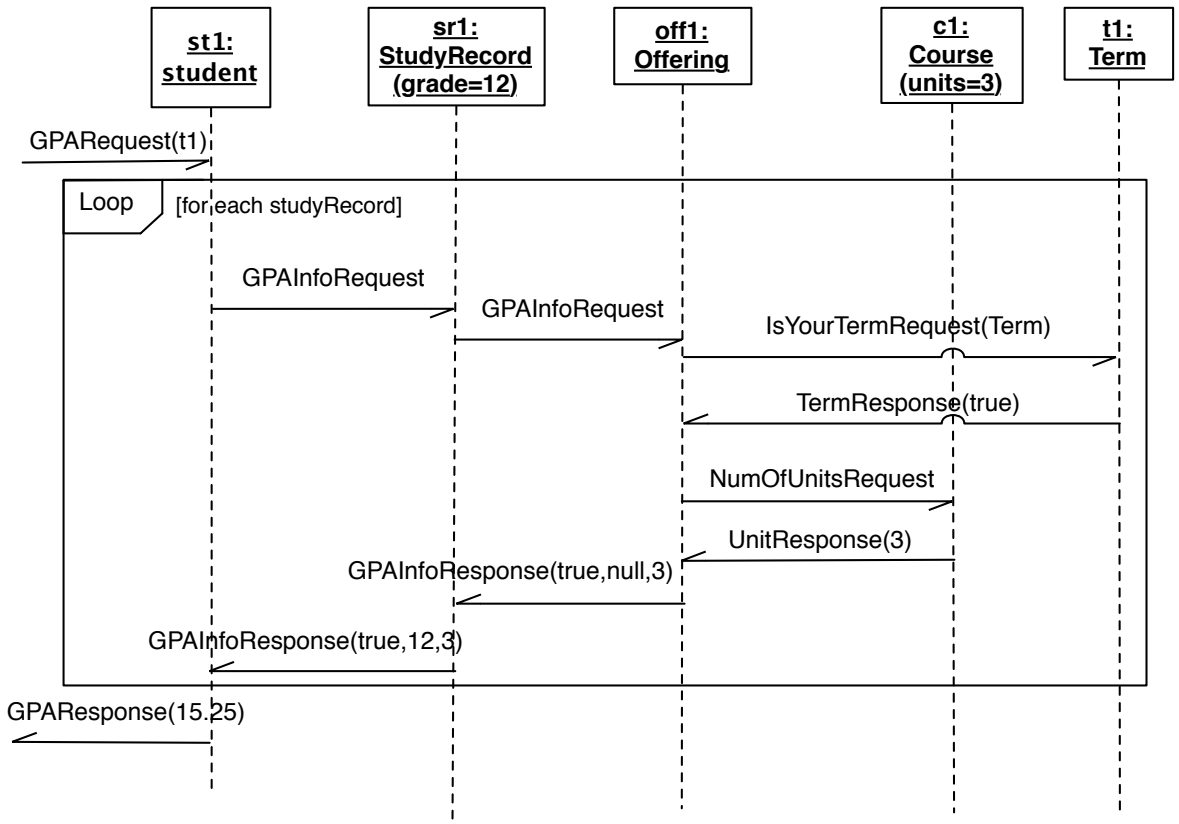
IsYourTermRequest(term: Term)

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 در خواست 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

```
class Student(
  var id: String,
  var name: String,
  var studyRecords: List[StudyRecord]) extends Actor {

  var weightedSumOfGrades = 0
  var sumOfUnits=0

  override def act() {
    loop {
      react {
       case GPARequest(term: Term) =>{
          val replies = for(sr <- studyRecords) yield {sr !! GPAInfoRequest(term)}
          for(i <- 0 until offerings.size) {
            GPAInfoResponse(isForTerm:Boolean, grade: Double, units:Int) = replies(i)
            if(isForTerm) {
              weightedSumOfGrades += units * grade
              sumOfUnits += sumOfUnits
            }
          }
          sender ! GPAResponse(weightedSumOfGrades / sumOfUnits)
       }
       case TakeCourseRequest(offering:Offering)=>
       ...
      }
    }
  }
}
```

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

```
class Offering(
  var id: String,
  var course: Course,
  var examDate: Date,
  var term: Term) extends Actor {
  override def act() {
    loop {
      react {
       case GPAInfoRequest(gpaTerm: Term) =>
         val termFuture = term !! IsYourTermRequest(gpaTerm)
         val courseFuture = course !! NumOfUnitsRequest
         sender ! GPAInfoResponse(termFuture(),null,courseFuture())
         }
      }
    }
  }
}
```

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.

```
class Term(
  var name: String,
  var startDate: Date,
  var offerings: List[Offering]) extends Actor {
  override def act() {
    loop {
      react {
       case IsYourTermRequest(gpaTerm) =>
         sender ! (gpaTerm.name == name)
      }
    }
  }
}
```

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

```
class Course(
  var id: String,
  var name: String,
  var units: Int,
  var preRequisites: List[Course]) extends Actor {
  override def act() {
    loop {
      react {
       case NumOfUnitsRequest =>
         sender ! units
      }
    }
  }
}
```

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

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 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.

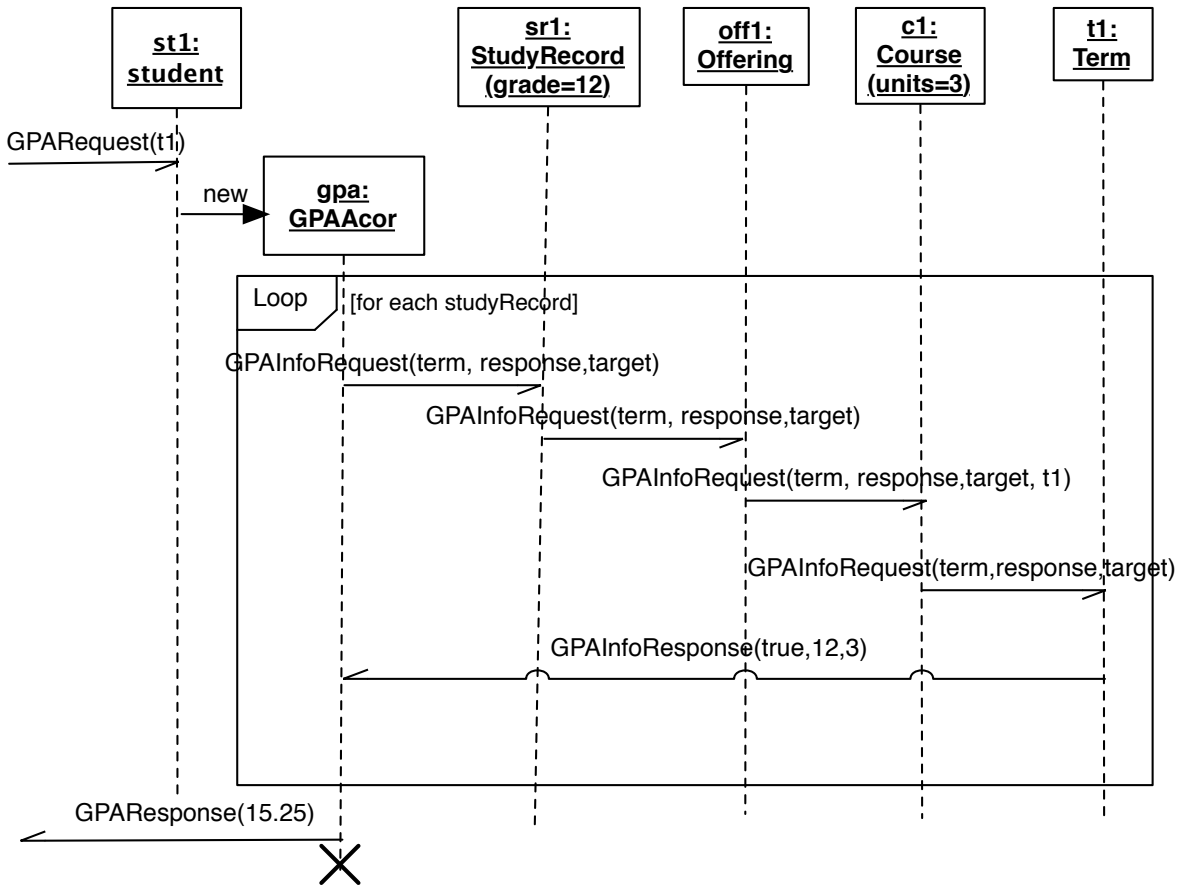
مورد دیگری که در رویکرد ۱ بررسی می‌کنیم عدم امکان پردازش درخواست‌های جدید در هنگام انتظار برای تکمیل اطلاعات مورد نیاز برای پاسخ به درخواست قبلی است. مثلا در مورد دانشجو این مورد باعث شد که در رویکرد ۱، دانشجو قبل از ارسال پاسخ درخواست معدل، درخواست دیگری را بررسی کند. در مورد دانشجو دلیل این پدیده این 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.

```
class GPAActor(
  val term: Term,
  val studyRecords: List[StudyRecord],
   val target: Actor) extends Actor {

  var processedMessages = 0
  var weightedSumOfGrades = 0
  var sumOfUnits = 0

  override def act() {
    for(sr <- studyRecords)
      sr ! GPAInfoRequest(term, this, )

    loop {
      react {
        case GPAInfoResponse(isForTerm:Boolean, grade: Double, units:Int) =>
          processMessage(isForTerm, grade, units)
      }
    }
  }

  def processMessage(isForTerm:Boolean, grade:Double, units:Int) {
    if(isForTerm) {
            weightedSumOfGrades += units * grade
            sumOfUnits += sumOfUnits
    }
    processedMessages ++
    if(processedMessages == studyRecords.size) {
      target ! GPAResponse(weightedSumOfGrades / sumOfUnits)
      exit
    }
  }
}
```

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

```
class Student(
  var id: String,
  var name: String,
  var studyRecords: List[StudyRecord]) extends Actor {

  override def act() {
    loop {
      react {
       case GPARequest(term: Term) =>
        val gpa = new GPAActor(term, studyRecords, sender)
        gpa.start
       case TakeCourseRequest(offering:Offering)=>
       ...
      }
    }
  }
}
```

شکل ۱۷.۴ : شبه‌کد طراحی اکتور دانشجو در رویکرد ۲.

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

[۲۸] در طراحی شیءگرای ترتیبی، مکانیزم کنترل برنامه فراخوانی متد است. با هر فراخوانی متد، منطق پیاده شده در متد اجرا می‌شود

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 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

و پس از بازگشت از متد، اجبارا کنترل برنامه به همان قسمتی که متد فراخوانی شده بود برمی‌گردد.

۴.۴ الگوهای طراحی استخراج شده و نکات مهم

در این بخش الگوهای طراحی و نکات مهمی که در طول انجام طراحی موارد کاربرد به دست آمده است گردآوری و ارائه شده است. نحوه‌ی تقسیم‌بندی موارد این بخش به این صورت است که ابتدا الگوهای کلی همکاری اکتورها برای پیاده‌سازی منطق دامنه برشمرده شده‌اند و برای هر مورد سعی شده تأثیر منطق دامنه در انتخاب الگو و نیز در نحوه‌ی پیاده‌سازی جزئیات الگو در نظر گرفته شود. در ادامه الگوها و نکته‌های مهم در طراحی پیغام‌ها ارائه شده‌اند. نهایتا نکات و تجربیاتی که در زمینه‌ی طراحی به روش انتقال ناهمگام و تفاوت‌های مهم آن با طراحی شیءگرای ترتیبی ارائه شده است.

۱.۴.۴ الگوهای کلی همکاری اکتورها

الگوی انشعاب و الحاق

- **نحوه‌ی پیاده‌سازی به روش تبادل ناهمگام پیغام:**

همان‌طور که از نام این الگو بر می‌آید پیاده‌سازی آن از دو بخش تشکیل شده است. برای عمل انشعاب یک اکتور به تعدادی اکتور دیگر که به آنها دسترسی دارد و یا خود آنها را ایجاد می‌کند پیغام‌هایی می‌فرستد. این اکتورها تمام عملیات لازم برای تهیه‌ی پاسخ را انجام داده و پیغام‌های پاسخ را برای اکتور اصلی ارسال می‌کنند. مرحله‌ی جمع‌آوری پیغام‌ها انشعاب نامیده می‌شود. اکتور اصلی این پیغام‌ها را دریافت کرده و محاسبات لازم را روی آنها انجام می‌دهد. و در پاسخ عملیات را به صورت پیغام ارسال می‌کند.

- **موارد استفاده**

این الگو برای حالاتی از منطق دامنه به کار می‌رود که مسئله از نوع تقسیم و حل است (رجوع کنید به بخش ۱.۳). مثالی از این کارکرد در مورد کاربرد محاسبه‌ی معدل (جدول ۱.۴) در بخش قبل مورد بررسی قرار گرفته است. در این مثال، اکتوری که وظیفه‌ی محاسبه‌ی معدل را بر عهده دارد به اکتورهای سابقه‌ی تحصیلی دانشجو دسترسی دارد و برای محاسبه‌ی معدل نیاز به اطلاعاتی دارد که این اکتورها به آن دسترسی دارند. استفاده از الگوی انشعاب و الحاق در این مثال به این صورت است که اکتور محاسبه‌ی معدل پیغام‌های درخواست اطلاعات نمره را برای تمام اکتورهای سابقه ارسال می‌کند (انشعاب)، این اکتورها با برقراری ارتباط با سایر اکتورها موجب می‌شوند اطلاعات لازم به صورت پیغام‌هایی برای اکتور محاسبه‌ی معدل ارسال شود. اکتور محاسبه‌ی معدل با گرفتن

پیغام‌ها (الحاق) عملیات مورد نیاز برای محاسبه‌ی معدل را انجام می‌دهد و معدل را به صورت پیغام برای اکتور مقصد ارسال می‌کند.

- **نکات مهم**

۱. مناسب بودن مسائل تقسیم و حل برای این الگو به این معنی نیست که نمی‌توان از روش دیگری این مسائل را حل کرد. به عنوان مثال، مورد کاربرد محاسبه‌ی معدل در رویکرد اول طراحی که در بخش ۲.۳.۴ توضیح داده شد، بدون استفاده از این الگو طراحی گردید. همان‌طور که در مقایسه‌ی دو رویکرد مذکور بیان شد، تفاوت استفاده و عدم استفاده از الگوی انشعاب و الحاق صرفا کیفیت طراحی می‌باشد و از نظر صحت عملکرد دو الگو یکسان هستند. بنابراین استفاده از این الگو بیشتر به تمرین در طراحی نیازمند است و صرفاً از روی منطق دامنه قابل تشخیص نیست.

۲. مورد دیگر این است که در بسیاری از موارد، استفاده از این الگو به ذهن برنامه‌نویس این طور القا می‌کند که اکتورهایی که انشعاب شده‌اند موظف به فرستادن نتیجه به اکتور اصلی هستند. باید دقت شود که استفاده از این الگو مستقل از این مورد است که نتیجه‌ی کارهای تقسیم شده به چه شکلی به دست اکتور اصلی می‌رسد. همان‌طور که در رویکرد دوم طراحی مورد کاربرد محاسبه‌ی معدل مشاهده شد، پاسخ اکتور محاسبه‌ی معدل می‌تواند از سوی اکتور ترم و یا اکتور درس ارسال شود.

۳. استفاده از این الگو صرفا با کشف مورد استفاده به اتمام نمی‌رسد. پس از تصمیم به استفاده از این الگو، تصمیمات دیگری در پاسخ به سؤالاتی از این قبیل باید اتخاذ شود: آیا اکتور جاری که منطق مطابق با الگو در آن کشف شده است باید به عنوان اکتوری که انشعاب در نظر گرفته شود یا اکتور دیگری به این منظور ایجاد شود؟ آیا اکتور الحاق و اکتور انشعاب یکسان باشند یا اکتور دیگری عمل انشعاب را انجام دهد؟ آیا نتیجه‌ی عملیات حاصل از الگو به اکتور جاری فرستاده شود یا مستقیماً به گیرنده‌ی دیگری ارسال شود؟

الگوی خط لوله

- **نحوه‌ی پیاده‌سازی به روش تبادل ناهمگام پیغام:**

همان‌طور که از نام این الگو بر می‌آید پیاده‌سازی آن از دو بخش تشکیل شده است. برای عمل انشعاب یک اکتور به تعدادی اکتور دیگر که به آنها دسترسی دارد و یا خود آنها را ایجاد می‌کند پیغام‌هایی می‌فرستد. این اکتورها

تمام عملیات لازم برای تهیه‌ی پاسخ را انجام داده و پیغام‌های پاسخ را برای اکتور اصلی ارسال می‌کنند. مرحله‌ی جمع‌آوری پیغام‌ها انشعاب نامیده می‌شود. اکتور اصلی این پیغام‌ها را دریافت کرده و محاسبات لازم را روی آنها انجام می‌دهد. و در پاسخ عملیات را به صورت پیغام ارسال می‌کند.

- **موارد استفاده**

این الگو برای حالاتی از منطق دامنه به کار می‌رود که مسئله از نوع تقسیم و حل است (رجوع کنید به بخش ۱.۳). مثالی از این کارکرد در مورد کاربرد محاسبه‌ی معدل (جدول ۱.۴) در بخش قبل مورد بررسی قرار گرفته است. در این مثال، اکتوری که وظیفه‌ی محاسبه‌ی معدل را بر عهده دارد به اکتورهای سابقه‌ی تحصیلی دانشجو دسترسی دارد و برای محاسبه‌ی معدل نیاز به اطلاعاتی دارد که این اکتورها به آن دسترسی دارند. استفاده از الگوی انشعاب و الحاق در این مثال به این صورت است که اکتور محاسبه‌ی معدل پیغام‌های درخواست اطلاعات نمره را برای تمام اکتورهای سابقه ارسال می‌کند (انشعاب)، این اکتورها با برقراری ارتباط با سایر اکتورها موجب می‌شوند اطلاعات لازم به صورت پیغام‌هایی برای اکتور محاسبه‌ی معدل ارسال شود. اکتور محاسبه‌ی معدل با گرفتن پیغام‌ها (الحاق) عملیات مورد نیاز برای محاسبه‌ی معدل را انجام می‌دهد و معدل را به صورت پیغام برای اکتور مقصد ارسال می‌کند.

- **نکات مهم**

۱. مناسب بودن مسائل تقسیم و حل برای این الگو به این معنی نیست که نمی‌توان از روش دیگری این مسائل را حل کرد. به عنوان مثال، مورد کاربرد محاسبه‌ی معدل در رویکرد اول طراحی که در بخش ۲.۳.۴ توضیح داده شد، بدون استفاده از این الگو طراحی گردید. همان‌طور که در مقایسه‌ی دو رویکرد مذکور بیان شد، تفاوت استفاده و عدم استفاده از الگوی انشعاب و الحاق صرفا کیفیت طراحی می‌باشد و از نظر صحت عملکرد دو الگو یکسان هستند. بنابراین استفاده از این الگو بیشتر به تمرین در طراحی نیازمند است و صرفاً از روی منطق دامنه قابل تشخیص نیست.

۲. مورد دیگر این است که در بسیاری از موارد، استفاده از این الگو به ذهن برنامه‌نویس این طور القا می‌کند که اکتورهایی که انشعاب شده‌اند موظف به فرستادن نتیجه به اکتور اصلی هستند. باید دقت شود که استفاده از این الگو مستقل از این مورد است که نتیجه‌ی کارهای تقسیم شده به چه شکلی به دست اکتور اصلی می‌رسد. همان‌طور که در رویکرد دوم طراحی مورد کاربرد محاسبه‌ی معدل مشاهده شد، پاسخ اکتور محاسبه‌ی معدل می‌تواند از سوی اکتور ترم و یا اکتور درس ارسال شود.

۳. استفاده از این الگو صرفا با کشف مورد استفاده به اتمام نمی‌رسد. پس از تصمیم به استفاده از این الگو، تصمیمات دیگری در پاسخ به سؤالاتی از این قبیل باید اتخاذ شود: آیا اکتور جاری که منطق مطابق با الگو در آن کشف شده است باید به عنوان اکتوری که انشعاب در نظر گرفته شود یا اکتور دیگری به این منظور ایجاد شود؟ آیا اکتور الحاق و اکتور انشعاب یکسان باشند یا اکتور دیگری عمل انشعاب را انجام دهد؟ آیا نتیجه‌ی عملیات حاصل از الگو به اکتور جاری فرستاده شود یا مستقیماً به گیرنده‌ی دیگری ارسال شود؟

فصل ۵

ارزیابی

در فصل قبل اجزای چهارچوب پیشنهادی این پژوهش به تفصیل تشریح شد و در م

۱.۵ روش ارزیابی

۲.۵ ارزیابی کارایی

سبیس

۳.۵ ارزیابی تغییرپذیری

سبیس

۱.۳.۵ بررسی معیارهای ایستا

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

۲.۳.۵ اعمال تغییرات

تغییر اول

۴.۵ نتایج ارزیابی

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

۱.۴.۵ تحلیل نتایج

با داشتن نتای

فصل ۶

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.

۱.۶ دستاوردهای این پژوهش

این پژوهش، چهارچوبی بدیع برای آزمون سیستم‌های نرم‌افزاری بر پسیستم واقعی استفاده می‌شود.

در واقع چهارچوب پیشنهاد شده تلاش می‌کند تا مجموعه‌ی به هم‌پیوسته‌ای از فعالیت‌ها برای آزمون را، از اولین مراحل طراحی تا نتیجه‌گیری از مجموعه‌ی آزمون‌ها، پیشنهاد کند. در زیر برخی از مهم‌ترین دستاوردهای هر یک از مراحل این کار آمده است:

۲.۶ کاستی‌های چهارچوب

چهارچوب پیشنهاد شده در این پژوهش دارای کاستی‌هایی نیز هست که کار بیشتری را می‌طلبد. در این بخش به طور فهرست‌وار به برخی از آن‌ها اشاره می‌کنیم:

۳.۶ جهت‌گیری‌های پژوهشی آینده

بهره می‌برند را نیز می‌توان به شکل زیر برشمرد:

پیوست آ

تطبیق نمادگذاری‌ها

متن برنامه‌ی طراحی شده به روش ارسال ناهمگام پیغام

سلام

متن برنامه‌ی طراحی شده به روش شیء‌گرا

تعریف ذکر شین گذار نمادین به طور ساده به این شکل است:

کتاب‌نامه

[1] J. pierre Briot, R. GUERRAOUI, K.-P. Löhr, and K. peter L, "Concurrency and distribution in object-oriented programming," tech. rep., 1998. 5, 9

[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. 5, 6

[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. 5, 8

[4] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass, 1986. 5, 6

[5] G. Agha and C. Hewitt, "Concurrent programming using actors," pp.37–53, 1987. 6

[6] G. Agha, "Concurrent object-oriented programming," *Commun. ACM*, vol.33, no.9, pp.125–141, 1990. 6, 19, 21

[7] R. K. Karmani and G. Agha, "Actors," in *Encyclopedia of Parallel Computing*, pp.1–11, 2011. 7, 21, 24

[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. 7, 8, 10, 13, 22

[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. 8

[10] W. Kim and G. Agha, "Efficient support of location transparency in concurrent object-oriented programming languages," in *SC*, 1995. 9

[11] P.-H. Chang and G. Agha, "Towards context-aware web applications," in *DAIS*, pp.239–252, 2007. 9

[12] V. A. Korthikanti and G. Agha, “Towards optimizing energy costs of algorithms for shared memory architectures,” in *SPAA*, pp.157–165, 2010. 9

[13] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, second ed. , 1996. 9

[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. 10

[15] E. A. Lee, “Overview of the ptolemy project,” Tech. Rep. UCB/ERL M03/25, University of California, Berkeley, 2003. 10

[16] C. A. Varela and G. Agha, “Programming dynamically reconfigurable open systems with salsa,” *SIGPLAN Notices*, vol.36, no.12, pp.20–34, 2001. 10

[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. 10

[18] M. Astley, “The actor foundry: A java-based actor programming environment,” Open Systems Laboratory, Uni- versity of Illinois at Urbana-Champaign, 1998-99. 10

[19] Microsoft Corporation, “Asynchronous agents library,” `http://msdn.microsoft.com/en-us/library/dd492627.aspx`. 10

[20] B. V. Martin Odersky, Lex Spoon. *Programming In Scala*. WALNUT CREEK, CALIFORNIA: artima, 2 ed. , 2010. 10

[21] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1996. 16

[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. 16

[23] John Ousterhout, “Why threads are a bad idea (for most purposes),” Invited talk at USENIX, January 1996. 16

[24] R. von Behren, J. Condit, and E. Brewer, “Why events are a bad idea (for high-concurrency servers),” in *IN HOTOS*, 2003. 16

[25] B. Chin and T. Millstein, “T.d.: Responders: Language support for interactive applications,” in *In: Proc. ECOOP*, pp.255–278, 2006. 16

[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>. 17

[27] T. H. Feng and E. A. Lee, “Scalable models using model transformation,” 2008. 20

[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. 21

[29] T. Papaioannou, “On the structuring of distributed systems : the argument for mobility.,” 2000. 21

[30] S. Frølund. *Coordinating distributed objects: an actor-based approach to synchronization*. Cambridge, MA, USA: MIT Press, 1996. 24

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
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

Integrating Functional and Structural Methods In Model-Based Testing

Abstract

Model-based testing (i.e. automatic test-case generation based on functional models of the system under test) is now widely in use as a solution to automatic software testing problem. The goal this testing method is to test complex systems (e.g. systems with concurrent behaviors). By the way, it exploits low-level notations (e.g. transition systems) to describe system specifications. Therefore, modeling some aspects of the system, such as input/output data values, may result in high-complexity of the resulted model or it may not possible at all. On the other hand, there are methods that focus on data-dependent systems. Hereby, they analyze the source code (in a white-box manner), instead if high-level behavioral models, to infer data dependencies and to define test data valuation method. In spite of their power in modeling data items, test behaviors in these methods should be designed manually and therefore, defining numerous and complex behaviors for the test process may lead to difficulties.

In this work, we introduce an integrated framework for modeling both the expected system behaviors and the input/output data structures, consistently. To this end, we have used UML language for modeling purposes. This enables us to describe systems that are both complex in behavior and the data. We have also developed a tool which automatically generates test-cases based on the defined UML models.

Keywords: *model-based testing, automatic test generation, test framework, testing data dependent systems, category partitioning methods.*

University of Tehran

School of Electrical and Compuer Engineering

Integrating Functional and Structural Methods In Model-Based Testing

by

Hamid Reza Asaadi

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

June 2010