دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

عنوان:

الگوها در سیستم های نهفته بی درنگ

نویسنده

علی محسنی نژاد

استاد

دکتر رامان رامسین

مرداد ۱۴۰۳

فهرست مطالب

۱ مقدمه ۵

۲ پیشینه پژوهش ۶
۱.۲ الگوهای طراحی برای دسترسی به سخت‌افزار ۶
۱.۱.۲ الگوی Hardware Proxy ۶
۲.۱.۲ الگوی Hardware Adapter ۷
۳.۱.۲ الگوی Mediator ۷
۴.۱.۲ الگوی Observer ۸
۵.۱.۲ الگوی Debouncing ۹
۶.۱.۲ الگوی Interrupt ۹
۷.۱.۲ الگوی Polling ۱۰
۲.۲ الگوهای طراحی برای همزمانی نهفته و مدیریت حافظه ۱۰
۱.۲.۲ الگوی Cyclic Executive ۱۰
۲.۲.۲ الگوی Static Priority ۱۱
۳.۲.۲ الگوی Critical Region ۱۱
۴.۲.۲ الگوی Guarded Call ۱۳
۵.۲.۲ الگوی Queuing ۱۳
۶.۲.۲ الگوی Rendezvous ۱۳
۷.۲.۲ الگوی Simultaneous Locking ۱۳
۸.۲.۲ الگوی Ordered Locking ۱۳
۳.۲ الگوهای طراحی برای ماشین‌های حالت ۱۳
۱.۳.۲ الگوی Single Event Receptor ۱۳
۲.۳.۲ الگوی Multiple Event Receptor ۱۳
۳.۳.۲ الگوی State Table ۱۳
۴.۳.۲ الگوی State ۱۴
۵.۳.۲ And States ۱۴
۶.۳.۲ الگوی Decomposed And State ۱۴
۴.۲ الگوهای امنیت و قابلیت اطمینان ۱۴
۱.۴.۲ الگوی One’s Complement ۱۴
۲.۴.۲ الگوی CRC ۱۴
۳.۴.۲ الگوی Smart Data ۱۴
۴.۴.۲ الگوی Channel ۱۴
۵.۴.۲ الگوی Protected Single Channel ۱۴
۶.۴.۲ الگوی Dual Channel ۱۴
۵.۲ الگوهای معماری زیربخش‌ها و اجزا ۱۴
۱.۵.۲ الگوی Layered ۱۵
۲.۵.۲ الگوی Five Layer ۱۵
۳.۵.۲ الگوی Microkernel ۱۵
۴.۵.۲ الگوی Channel ۱۵
۵.۵.۲ الگوی Recursive Containment ۱۵
۶.۵.۲ الگوی Hierarchical Control ۱۵
۷.۵.۲ الگوی Virtual Machine ۱۶
۸.۵.۲ معماری Component-Based ۱۶
۹.۵.۲ الگوی ROOM ۱۶
۶.۲ الگوهای معماری هم‌زمانی ۱۶

۱.۶.۲ الگوی Message Queuing ۱۶
۲.۶.۲ الگوی Interrupt ۱۷
۳.۶.۲ الگوی Guarded Call ۱۷
۴.۶.۲ الگوی Rendezvous ۱۷
۵.۶.۲ الگوی Cyclic Execution ۱۷
۶.۶.۲ الگوی Round Robin ۱۷
۷.۶.۲ الگوی Static Priority ۱۷
۸.۶.۲ الگوی Dynamic Priority ۱۸
۷.۲ الگوهای معماری حافظه ۱۸
۱.۷.۲ الگوی Static Allocation ۱۸
۲.۷.۲ الگوی Pool Allocation ۱۸
۳.۷.۲ الگوی Fixed Sized Buffer ۱۸
۴.۷.۲ الگوی Smart Pointer ۱۸
۵.۷.۲ الگوی Garbage Collection ۱۹
۶.۷.۲ الگوی Garbage Compactor ۱۹
۸.۲ الگوهای معماری منابع ۱۹
۱.۸.۲ الگوی Critical Section ۱۹
۲.۸.۲ الگوی Priority Inheritance ۱۹
۳.۸.۲ الگوی Highest Locker ۱۹
۴.۸.۲ الگوی Priority Ceiling ۲۰
۵.۸.۲ الگوی Simultaneous Locking ۲۰
۶.۸.۲ الگوی Ordered Locking ۲۲
۹.۲ الگوهای معماری توزیع ۲۲
۱.۹.۲ الگوی Shared Memory ۲۲
۲.۹.۲ الگوی Remote Method Call ۲۲
۳.۹.۲ الگوی Observer ۲۲
۴.۹.۲ الگوی Data Bus ۲۲
۵.۹.۲ الگوی Proxy ۲۲
۶.۹.۲ الگوی Broker ۲۲
۱۰.۲ الگوهای معماری امنیت و قابلیت اطمینان ۲۲
۱.۱۰.۲ الگوی Protected Single Channel ۲۲
۲.۱۰.۲ الگوی Homogeneous Redundancy ۲۲
۳.۱۰.۲ الگوی Triple Modular Redundancy ۲۲
۴.۱۰.۲ الگوی Heterogeneous Redundancy ۲۲
۵.۱۰.۲ الگوی Monitor-Actuator ۲۲
۶.۱۰.۲ الگوی Sanity Check ۲۲
۷.۱۰.۲ الگوی Watchdog ۲۲
۸.۱۰.۲ الگوی Safety Executive ۲۲
۱۱.۲ الگوهای سخت‌افزاری برای سیستم‌های Safety-Critical ۲۲
۱.۱۱.۲ الگوی Homogeneous Duplex ۲۲
۲.۱۱.۲ الگوی Heterogeneous Duplex ۲۲
۳.۱۱.۲ الگوی Triple Modular Redundancy ۲۲
۴.۱۱.۲ الگوی M-Out-Of-N ۲۲
۵.۱۱.۲ الگوی Monitor-Actuator ۲۲
۶.۱۱.۲ الگوی Sanity Check ۲۲
۷.۱۱.۲ الگوی Watchdog ۲۲
۸.۱۱.۲ الگوی Safety Executive ۲۲

۱۲.۲ الگوهای نرم‌افزاری برای سیستم‌های Safety-Critical ۲۲
۱.۱۲.۲ الگوی N-Version Programming ۲۲
۲.۱۲.۲ الگوی Recovery Block ۲۲
۳.۱۲.۲ الگوی Acceptance Voting ۲۲
۴.۱۲.۲ الگوی N-Self Checking Programming ۲۲
۵.۱۲.۲ الگوی Recovery Block with Backup Voting ۲۲
۱۳.۲ الگوهای ترکیبی سخت‌افزار و نرم‌افزار برای سیستم‌های Safety-Critical ۲۲
۱.۱۳.۲ الگوی Protected Single Channel ۲۲
۲.۱۳.۲ الگوی 3-Level Safety Monitoring ۲۲

۳ تحلیل ۲۳

۴ مراجع ۲۴

۱ مقدمه

این گزارش به طور مفصل به توضیح الگوهای معرفی‌شده در مقالات و کتب مختلف در حوزه سیستم‌های نهفته و بی‌درنگ می‌پردازد. برای درک عمیق‌تر این الگوها، باید ابتدا مشخص شود که منظور از سیستم‌های نهفته بی‌درنگ چیست. سیستم‌های نهفته در بخش‌های زیادی از زندگی روزمره وجود دارند؛ به طور مثال سیستم‌های رادیویی، سیستم‌های ناوبری، سیستم‌های تصویربرداری. به طور کلی یک سیستم نهفته را می‌توان اینگونه تعریف کرد: «یک سیستم کامپیوتری که به طور مشخص برای انجام یک کار در دنیای واقعی تخصیص داده‌شده و هدف آن ایجاد یک محیط کامپیوتری با کاربری عام نیست» [۱]. یک دسته مهم از سیستم‌های نهفته، سیستم‌های بی‌درنگ هستند. «سیستم‌های بی‌درنگ، سیستم‌هایی هستند که در آن‌ها قیدهای زمانی مشخص باید برآورده شوند تا سیستم بتواند به درستی کار کند» [۱].

حال که مفهوم سیستم‌های نهفته بی‌درنگ را دریافتیم، باید تعریفی از الگو در این سیستم‌ها ارائه دهیم. منابع متنوع تعاریف متفاوتی از الگوها ارائه کرده‌اند و بسیاری از آن‌ها این تعریف را به الگوهای طراحی محدود می‌کنند [۱]. هدف این گزارش تقسیم‌بندی الگوهای نرم‌افزاری به طور کلی نیست و صرفا می‌خواهیم الگوهای مورد استفاده در سیستم‌های نهفته و بی‌درنگ را بررسی کنیم. Zalewski [۲] می‌گوید: «یک الگو یک مدل یا یک قالب نرم‌افزاری است که به فرایند ایجاد نرم‌افزار کمک می‌کند.» این تعریف در عین سادگی، جامع است؛ به طوری که الگوهای طراحی، معماری و فرایندی را در خود شامل می‌شود. با این حال این مقاله نیز مانند بسیاری از دیگر مقالات، تعریف جدیدی از الگوها در سیستم‌های نهفته بی‌درنگ ارائه نکرده‌اند و برای تعریف آن به تعریف Gamma و دیگران [۳] از الگوهای طراحی ارجاع داده‌اند.

گزارش پیش رو ابتدا در فصل ۲ به مطالعه کارهای پیشین در حوزه الگوهای سیستم‌های نهفته بی‌درنگ می‌پردازد. ساختار ارائه‌شده در این بخش به صورت خطی، کتاب‌ها و مقالات بیان شده را بررسی کرده و الگوهای بیان‌شده از طرف ایشان را با همان ساختار و دسته‌بندی مورد نظر آن منبع ذکر کرده‌است.

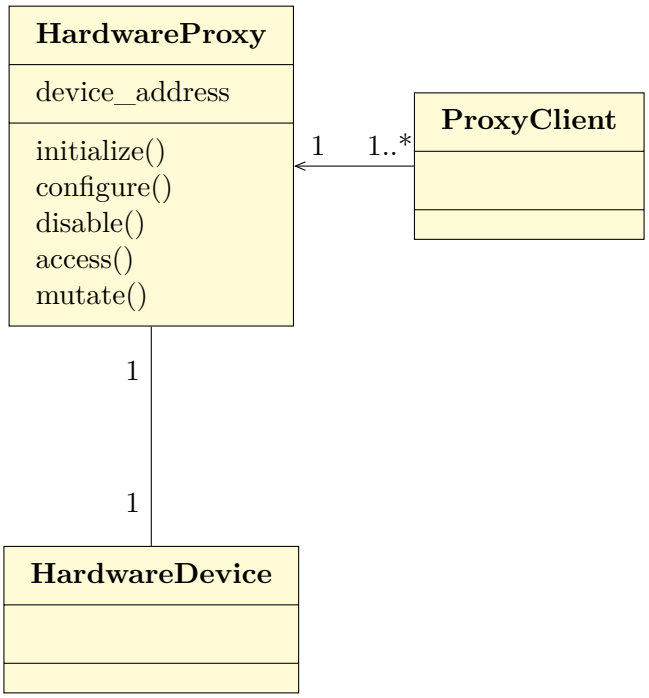
۲ پیشینه پژوهش

۱.۲ الگوهای طراحی برای دسترسی به سخت‌افزار

نرم‌افزارهای نهفته بر روی یک بستر سخت‌افزاری مستقر می‌شوند و معمولا بسیاری از قابلیت‌های آن‌ها ملزم به ارتباط با سخت‌افزار می‌شود. به همین دلیل Douglass [۱] یک دسته از الگوها را با عنوان الگوهای دسترسی به سخت‌افزار معرفی می‌کند.

۱.۱.۲ الگوی **Hardware Proxy**

این الگو با ایجاد یک رابط روی یک جزء سخت‌افزاری، یک دسترسی مستقل از پیچیدگی‌های اتصال به سخت‌افزار برای کلاینت ایجاد می‌کند. این الگو با معرفی یک کلاس به نام پروکسی بین سخت‌افزار و کلاینت، باعث می‌شود که تمامی عملیات وابسته به سخت‌افزار در پروکسی انجام شود و در صورت تغییر در سخت‌افزار، هیچ تغییری به کلاینت تحمیل نشود. در این الگو بر روی یک جزء سخت‌افزاری، یک پروکسی قرار گرفته و کلاینتان متعدد می‌توانند از آن سرویس بگیرند. لازم به ذکر است که ارتباط پروکسی و سخت‌افزار بر پایه یک «رابط قابل آدرس‌دهی توسط نرم‌افزار» است. دیاگرام کلاس این الگو در شکل ۱ رسم شده‌است.



شکل ۱: دیاگرام کلاس Hardware Proxy

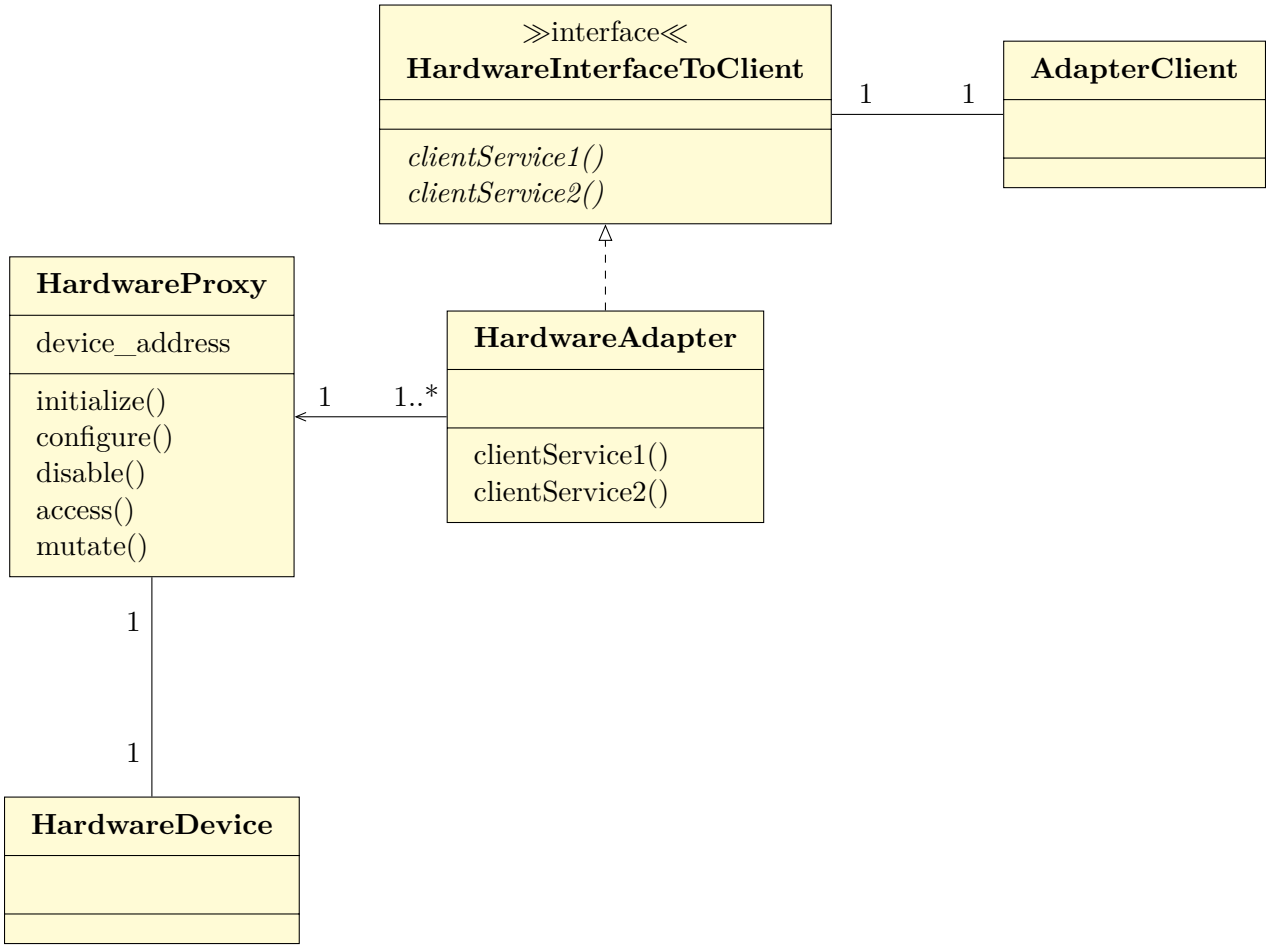
همانطور که در شکل ۱ دیده می‌شود، کلاس پروکسی توابع مشخصی را در اختیار کلاینت‌ها قرار می‌دهد[۱]. توضیحات مربوط به هر یک از توابع کلاس پروکسی در شکل زیر داده شده‌است:

- initialize: این تابع برای آماده‌سازی اولیه ارتباط با سخت‌افزار استفاده می‌شود و معمولا تنها یک بار صدا زده می‌شود.
- configure: این تابع برای ارسال تنظیمات برای سخت‌افزار استفاده می‌شود. معمولا باید در سخت‌افزار تنظیماتی قرار داده‌شود که آن را قابل استفاده کند.
- disable: این تابع برای غیرفعال‌کردن سخت‌افزار به صورت امن استفاده می‌شود.
- access: این تابع برای دریافت اطلاعات از طرف سخت‌افزار استفاده می‌شود.
- mutate: این تابع برای فرستادن اطلاعات به سمت سخت‌افزار استفاده می‌شود.

[۱] توابع دیگری نیز در [۱] گفته‌شده ولی اینجا تنها توابع public کلاس پروکسی را بررسی می‌کنیم.

۲.۱.۲ الگوی **Hardware Adapter**

این الگو مشابه الگوی Adapter که Gamma و دیگران [۳] معرفی کرده‌اند تعریف شده. استفاده از این الگو این اجازه را می‌دهد که کلاینتی که انتظار یک رابط خاص با سخت‌افزار را دارد، بتواند با سخت‌افزارهای مختلف بدون این‌که متوجه تفاوت‌های آن‌ها شود ارتباط بگیرد. این الگو روی ساختار الگوی Hardware Proxy بنا شده‌است و دیاگرام کلاس آن در شکل ۲ ترسیم شده‌است.



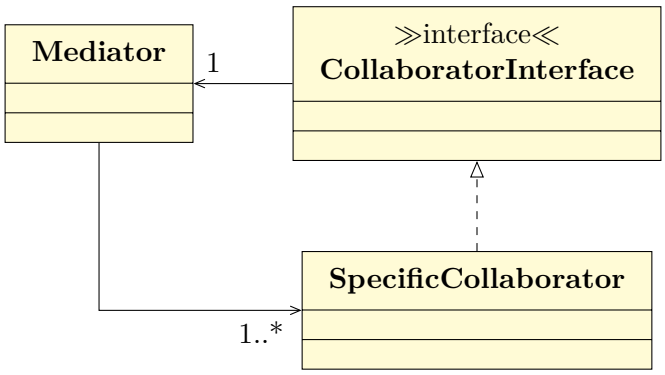
شکل ۲: دیاگرام کلاس Hardware Adapter

همانطور که در شکل ۲ دیده می‌شود، کلاس کلاینت سرویس‌های مورد انتظار خود را از رابط HardwareInterfaceToClient انتظار دارد. در این ساختار، کلاس آداپتور، سرویس‌های مورد انتظار کلاینت را به سرویس‌های ارائه‌شده از طرف سخت‌افزار ترجمه می‌کند. این کار اجازه می‌دهد که در صورت تغییر سخت‌افزار (و متناظرا پروکسی)، تنها با ایجاد پیاده‌سازی جدید برای رابط آداپتور، نیازی به تغییر در کلاینت نباشد.

۳.۱.۲ الگوی **Mediator**

این الگو با معرفی یک کلاس میانجی‌گر بین چند کلاس همکار، کمک می‌کند که چند سخت‌افزار را با هم مدیریت کند. ساختار این الگو در شکل ۳ ترسیم شده‌است.

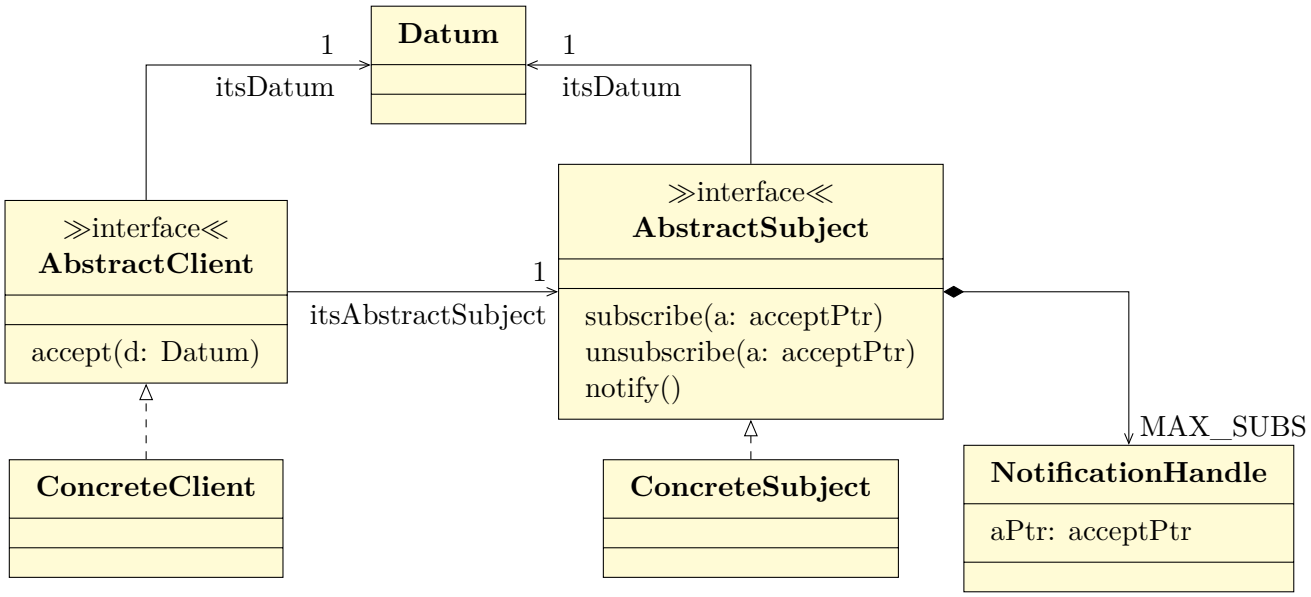
همانطور که در شکل مشخص است، کلاس میانجی با هر یک از کلاس‌های همکار ارتباط دارد. این ارتباط به این شکل است که کلاس میانجی تمامی پیاده‌سازی‌های رابط همکار را می‌شناسد و با آن‌ها ارتباط دارد. این کلاس‌ها خودشان نیز همانطور که نشان‌داده شده، میانجی را می‌شناسند و با آن ارتباط دارند. هر یک از کلاس‌های همکار، با سخت‌افزار در ارتباط هستند و حتی می‌توانند خود یک پروکسی باشند (الگوی Hardware Proxy). ولی به هر صورت در این الگو برای ارتباط با یکدیگر، باید برای میانجی سیگنال بفرستند و میانجی وظیفه ارتباطات بین همکارها را دارد (با ایجاد ارتباط غیر مستقیم). به طور کلی فرایندهایی که در آن استفاده از چند سخت‌افزار و نیاز است، توسط میانجی کنترل می‌شود.



شکل ۳: دیاگرام کلاس Mediator

۴.۱.۲ الگوی Observer

یکی از پرکاربردترین الگوها در حوزه سیستم‌های نهفته، الگوی Observer است. این الگو به شیءهای برنامه این اجازه را می‌دهد که به یک شیء دیگر برای دریافت اطلاعات گوش دهند. این به این معنی است که اگر یک کلاینت به دنبال دریافت داده از یک سرور است، به جای این که هر دفعه درخواست دریافت داده‌ها را برای سرور بفرستد، برای آن سرور درخواست عضویت فرستاده و سرور هرگاه که داده‌های جدید در دسترس بودند، آن‌ها را برای کلاینت‌های عضوشده بفرستد. یکی از مهم‌ترین کاربردهای این الگو در دریافت داده‌ها از سنسورها است. یکی از قابلیت‌های خوب این الگو این است که کلاینت‌ها می‌توانند در زمان اجرای برنامه عضویت خود را قطع یا ایجاد کنند. در شکل ۴ دیاگرام کلاس این الگو را می‌بینیم.



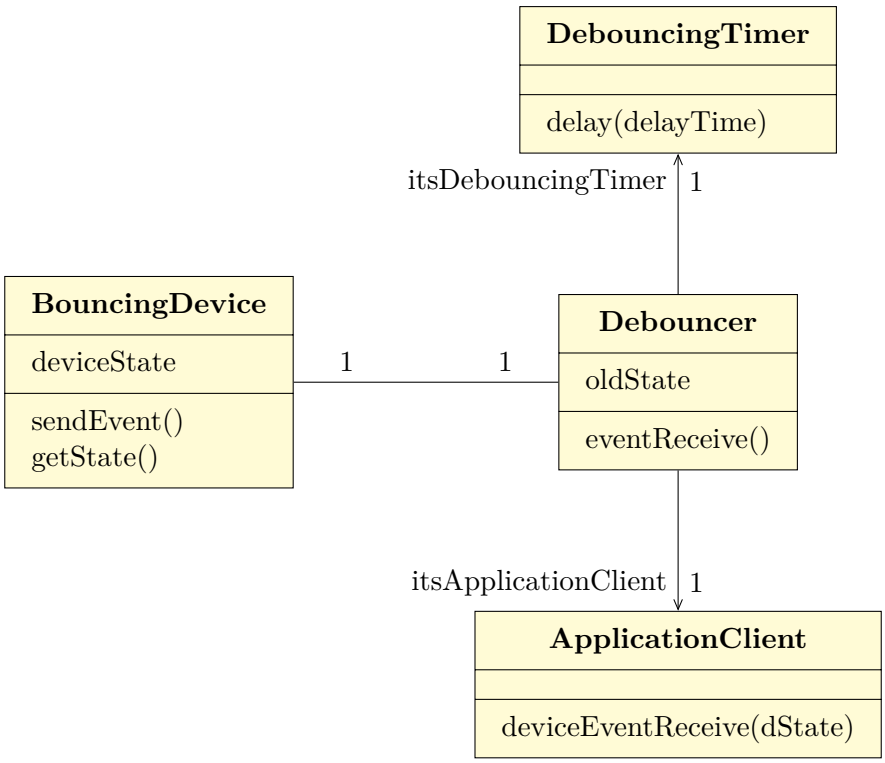
شکل ۴: دیاگرام کلاس Observer

در این ساختار کلاینت‌ها با فرستادن یک اشاره‌گر به کلاس سابجکت، درخواست عضویت برای سرویس می‌فرستند. کلاس سابجکت نیز با ذخیره‌کردن اشاره‌گرهای مختلف از طرف کلاینت‌ها زمانی که داده جدید آماده می‌شود، با فراخوانی تابع notify، تابع accept که اشاره‌گرهایش را در NotificationHandle قرار داده، با پاس دادن ورودی به فرمت Datum صدا می‌زند. اینگونه این داده برای تمامی کلاینت‌های عضو سرویس فرستاده می‌شود. کلاینت‌ها می‌توانند در حین اجرای برنامه، عضویت خود برای سرویس را لغو کنند. دقت شود که خود کلاس‌های سابجکت معمولا از نوع پروکسی هستند (الگوی Hardware Proxy).

۵.۱.۲ الگوی **Debouncing**

در سخت‌افزار بسیاری از ورودی‌ها به صورت دکمه‌ها و سوییچ‌هایی هستند که بر اثر ایجاد اتصال دو فلز با یکدیگر، باعث فعال‌شدن یک پایه شده و آغازگر یک عملیات در نرم‌افزار نهفته هستند. اتصال این دو فلز با یکدیگر دارای تعدادی حالت میانی است. به این صورت که اتصال با کمی لرزش همراه بوده و اتصال برای چند میلی‌ثانیه چند بار قطع و وصل می‌شود. این قطع و وصل شدن، باعث می‌شود که نتوانیم حالت فعلی سخت‌افزار را به درستی در نرم‌افزار ضبط کنیم.

این الگو به ما کمک می‌کند که با صبرکردن برای یک مدت کوتاه، مقدار ورودی را زمانی که پایدار شده‌است بخوانیم. با این کار دغدغه معتبر بودن مقدار خوانده‌شده را در کلاینت نخواهیم داشت. دیاگرام کلاسی این الگو در شکل ۵ نمایش داده شده‌است.



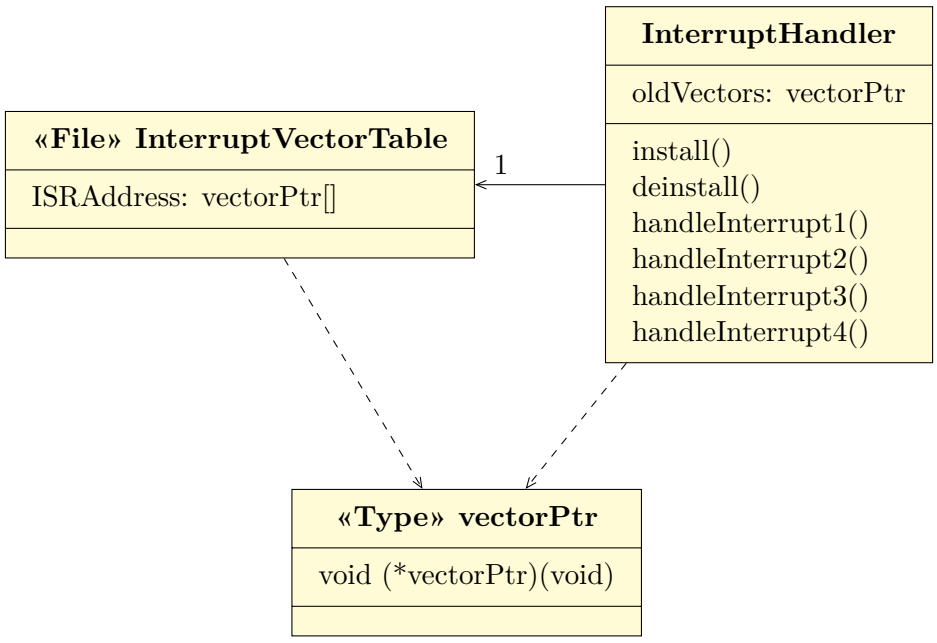
شکل ۵: دیاگرام کلاس Debouncing

در این ساختار، BouncingDevice همان سخت‌افزار مورد بررسی است. تابع sendEvent می‌تواند یک نوع اینتراپت سخت‌افزاری باشد که نرم‌افزار را از تغییر در سخت‌افزار باخبر می‌سازد و getState می‌تواند یک عملیات خواندن از حافظه باشد. کلاس Debouncer وظیفه ارائه حالت پایدار سخت‌افزار به کلاینت را دارد. این کار با استفاده از یک کلاس زمان‌سنج انجام می‌شود که با ایجاد یک تاخیر نرم‌افزاری تا پایدارشدن شرایط خروجی سخت‌افزار، خواندن حالت سخت‌افزار را به تعویق می‌اندازد.

۶.۱.۲ الگوی **Interrupt**

یکی از واحدهای مهم در سیستم‌های سخت‌افزاری، واحد Interrupt است. Interrupt برای هندل‌کردن وقایعی است که توسط سخت‌افزار جرقه زده می‌شوند. زمانی که یک interrupt رخ می‌دهد، نرم‌افزار فرایند کاری اصلی خود را متوقف کرده و یک فرایند رسیدگی به interrupt اتفاق‌افتاده آغاز می‌شود. با انجام این فرایند و رسیدگی به interrupt، فرایند اصلی نرم‌افزار دوباره از سر گرفته می‌شود. ساختار این الگو در شکل ۶ نمایش داده شده‌است.

در این الگو، کلاس InterruptHandler کار اصلی را انجام می‌دهد. این کلاس دارای بردار Interruptهاست. با فراخوانی تابع install می‌توان این بردار را با یک بردار جدید جایگزین‌کرد. این بردار در اصل تعدادی اشاره‌گر به توابعی است که در صورت بروز Interrupt باید فراخوانی شوند. با تابع deinstall نیز می‌توان بردار را به حالت قبلی برگرداند. فایل InterruptVectorTable شامل یک لیست از اشاره‌گرها به توابع Interrupt Service Routine است. و vectorPtr صرفا یک نوع اشاره‌گر به تابع است که از نوع آن در InterruptVectorTable استفاده شده‌است.



شکل ۶: دیاگرام کلاس Interrupt

۷.۱.۲ الگوی Polling

این الگو یک روش دیگر برای دریافت داده‌ها از سنسورها است و زمانی استفاده می‌شود که استفاده از الگوی Interrupt ممکن نیست یا این که داده‌هایی که می‌خواهیم ضبط کنیم آن قدر اضطراری نیستند و می‌توان برای دریافت آن‌ها صبر کرد. عملکرد این الگو به این صورت است که با سرکشی‌کردن به صورت دوره‌ای داده‌ها را دریافت می‌کنیم. حال این الگو در دو شکل بیان می‌شود: سرکشی داده‌ها به صورت دوره‌ای و به صورت فرصتی. در نوع اول با استفاده از یک تایمر، در زمان‌های مشخصی، برای دریافت داده‌های جدید سرکشی می‌کنیم که ساختار کلاسی آن نیز در شکل ۷ نشان داده شده‌است. در نوع دوم زمانی عمل سرکشی را انجام می‌دهیم که برای سیستم ممکن باشد و قیدهای زمانی سیستم به ما این اجاره را بدهد. ساختار کلاسی این نوع نیز در شکل ۸ رسم شده‌است.

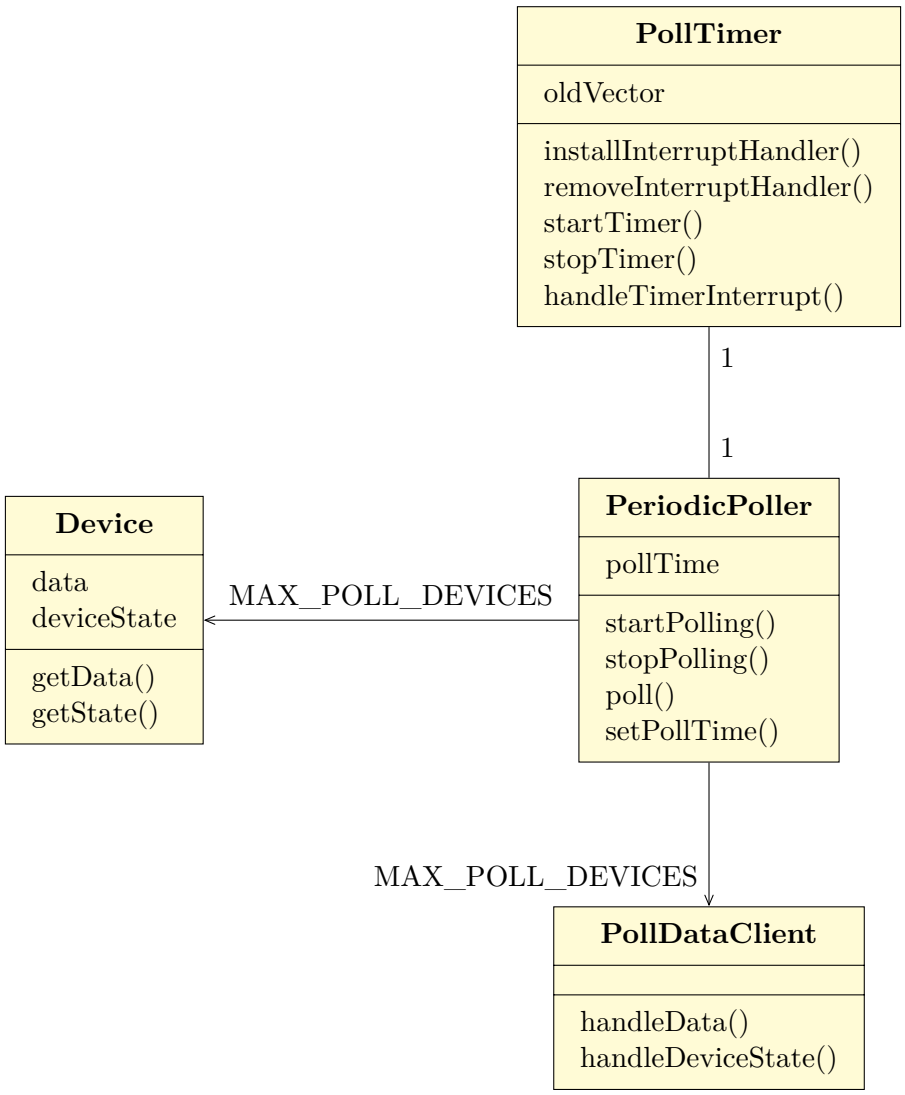
در این الگو کلاس Device همان سخت‌افزار/حافظه/... هست که می‌خواهیم داده‌هایش را دریافت کنیم. کلاس PollDataClient نیز کلاینتی است که می‌خواهد داده‌های Device را دریافت کند. در ساختار شکل ۷، کلاس PeriodicPoller با سرکشی از Device داده‌های آن را دریافت می‌کند. این سرکشی زمانی انجام می‌شود که کلاس PollTimer تابع poll را از PeriodicPoller صدا بزند. زمانی که فرمان startPolling به بیاید، تایمر کار خود را شروع می‌کند و در هر Interruptی که تایمر می‌خورد، تابع poll را صدا می‌زند. با فراخوانی تابع stopPolling، تایمر متوقف می‌شود. در ساختار ۸ تفاوت در این است که تابع poll زمانی صدا زده می‌شود که کلاس ApplicationProcessingElement بخواهد. یعنی زمانی که این کلاس در فرایندهای خود لازم می‌بیند که لازم است داده‌های جدید از Device گرفته‌شود، این تابع صدا زده می‌شود.

۲.۲ الگوهای طراحی برای همزمانی نهفته و مدیریت حافظه

در بسیاری از مواقع در سیستم‌های نهفته لازم است که فعالیت‌های متنوع به صورت همزمان انجام شوند. به همین دلیل Douglass [۱] یک دسته از الگوها به نام الگوهای برنامه‌ریزی را معرفی کرده‌است.

۱.۲.۲ الگوی Cyclic Executive

این الگو یکی از ساده‌ترین روش‌های زمانبندی در سیستم‌ها است. در این روش، هر تسک شانس مساوی برای اجرا شدن دارد و تمامی تسک‌ها در یک حلقه بی‌نهایت به صورت نوبتی جلو می‌روند. این الگو در دو موقعیت مشخص کاربرد دارد. موقعیت اول زمانی است که سیستم مورد بررسی یک سیستم نهفته بسیار کوچک است و می‌خواهیم بدون نیاز به الگوریتم‌های پیچیده زمانبندی به یک ساختار شبه‌هم‌زمان برسیم. موقعیت دوم زمانی است که سیستم مورد بررسی یک سیستم بسیار امن است و می‌خواهیم به طور قطع از انجام درست فرایند برنامه‌ریزی برای تسک‌ها و تحقق ددلاین‌ها مطمئن باشیم. ساختار کلاسی این الگو در شکل ۹ رسم شده‌است.



شکل ۷: دیاگرام کلاس Periodic Polling

کلاس CyclicExecutive با داشتن یک حلقه تکرار همیشگی، تابع run را از هر یک از AbstractCEThreadهایی که وجود دارد صدا می‌زند. [۲]

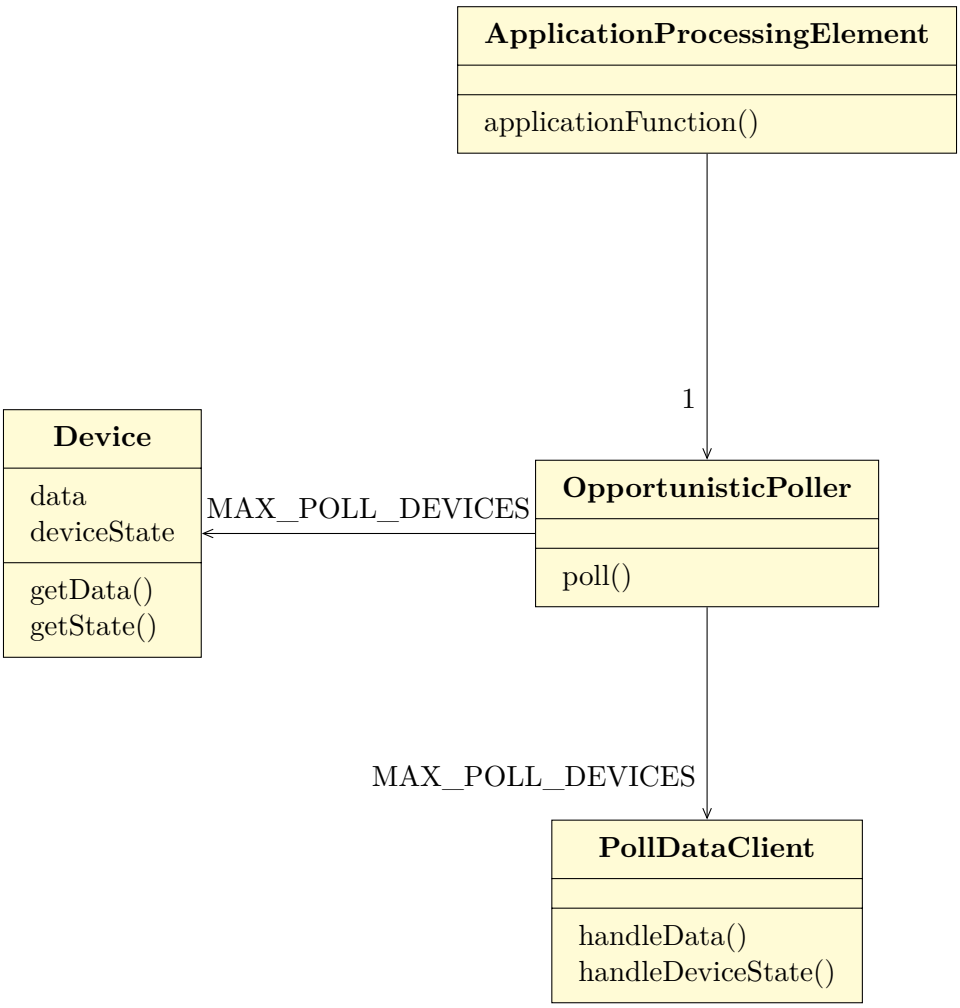
۲.۲.۲ الگوی **Static Priority**

این یکی از پرکاربردترین الگوهای برنامه‌ریزی در سیستم‌های نهفته بی‌درنگ است. این الگو به ما این قدرت را می‌دهد تا بتوانیم با استفاده از یک سیستم اولویت‌دهی به تسک‌ها، آن‌ها را انجام دهیم. در این سیستم فرض می‌شود که همه تسک‌ها از نوع سنکرون هستند و آن‌ها را بر اساس زمان ددلاینشان اولویت‌دهی می‌کنیم. به این شکل که تسک با نزدیک‌ترین ددلاین بالاترین اولویت را داشته‌باشد. این الگو نسبت به الگوی Cyclic Executive پیچیده‌تر بوده و هدف استفاده از آن، اولویت‌دهی به تسک‌های ضروری‌تر است.

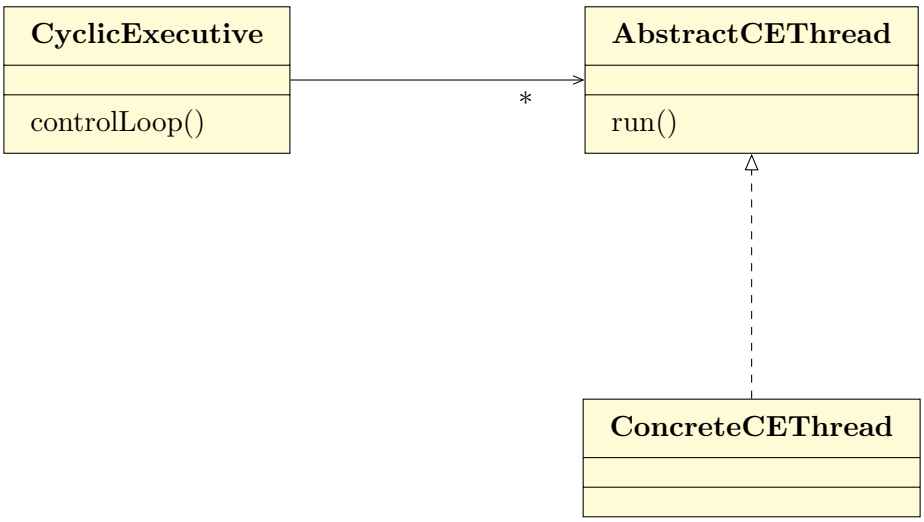
۳.۲.۲ الگوی **Critical Region**

این الگو برای زمانی استفاده می‌شود که می‌خواهیم یک تسک به خصوص بدون مزاحمت کار خود را به پایان برساند. این عملیات به این صورت است که زمانی که این تسک به خصوص انجام می‌شود، فرایند سوییچ‌کردن بین تسک‌ها را متوقف می‌کنیم تا زمانی که این تسک به پایان برسد. سپس دوباره فرایند زمان‌بندی تسک‌ها و سوییچ‌کردن بین آن‌ها به حالت عادی بازمی‌گردد. استفاده از این الگو معمولا در دو

[۲] در [۱]، یک کلاس دیگر نیز با نام CycleTimer وجود دارد. اما به دلیل کاربر کم، در اینجا درباره آن بحثی نمی‌کنیم.



شکل ۸: دیاگرام کلاس Opportunistic Polling



شکل ۹: دیاگرام کلاس Cyclic Executive

سناریو انجام می‌شود. اول، زمانی که تسک خاصی می‌خواهد از منبعی استفاده کند که تنها یک تسک باید به آن در یک لحظه دسترسی داشته‌باشد؛ در این صورت باید تا زمانی که این منبع در دسترس این تسک قرار گرفته‌است، از سوییچ‌کردن بین تسک‌ها خودداری کنیم. دوم،

زمانی که می‌خواهیم یک تسک به خصوص کار خود را در کوتاه‌ترین زمان ممکن انجام دهد؛ در این صورت نیز باید سوییچ‌کردن بین تسک‌ها را در زمان انجام این تسک متوقف کنیم.

۴.۲.۲ الگوی **Guarded Call**

این الگو با سری‌سازی دسترسی تسک‌ها به یک سرویس خاص، از استفاده همزمان آن جلوگیری می‌کند و جلوی تداخل‌های احتمالی را می‌گیرد. این الگو با استفاده از Semaphore این کار را انجام می‌دهد.

۵.۲.۲ الگوی **Queuing**

این الگو با بهره‌گیری از یک سیستم FIFO، می‌تواند بین تسک‌ها و رشته‌های مختلف برنامه پیام رد و بدل کند. استفاده از این سیستم برای تسک‌های آسنکرون بسیار ایده‌آل است. یکی دیگر از کاربردهای این الگو، در به اشتراک‌گذاری یک منبع مشترک بین تسک‌ها است. این الگو با ارسال داده‌ها از یک منبع به صورت pass by value مانع آلوده‌شدن منبع اصلی می‌شود و از Race جلوگیری می‌کند. مشکل این الگو این است که پیامی که از یک تسک به دیگری می‌رود، در همان لحظه پردازش نمی‌شود و باید تا فرارسیدن نوبت آن در صف صبر کند.

۶.۲.۲ الگوی **Rendezvous**

این الگو زمانی کاربرد دارد که شروط لازم برای سنکرون‌شدن تسک‌ها با یکدیگر پیچیده باشد. در این صورت الگوی Guarded Call و الگوی Queuing نمی‌توانند موثر واقع شوند و باید از الگوی Rendezvous استفاده کرد. این الگو با استفاده از یک شیء مجزا برای تحقق بخشیدن به سنکرون‌سازی تسک‌ها، تعدادی شرط تعریف می‌کند که با انجام آن‌ها، تسک‌ها سنکرون‌شده و آزاد می‌شوند. این کار به این صورت انجام می‌شود که هر یک از تسک‌ها خود را پیش شیء Rendezvous رجیسترکرده و تا زمانی که این کلاس تصمیم بگیرد متوقف می‌شوند.

۷.۲.۲ الگوی **Simultaneous Locking**

این الگو با هدف جلوگیری از Deadlock به وجود آمده‌است. این کار را به این روش انجام می‌دهد که اجازه نمی‌دهد هیچ تسکی یک منبع را زمانی قفل کند که خود منتظر آزادشدن یک منبع دیگر است.

۸.۲.۲ الگوی **Ordered Locking**

این الگو نیز برای جلوگیری از بروز Deadlock استفاده می‌شود. روش جلوگیری به این شکل است که منابع را به ترتیبی مرتب می‌کنیم و کلاینت‌های این منابع را مجبور می‌کنیم که این منابع را به همین ترتیب قفل و رها کنند. این کار جلوی صبرکردن چرخه‌ای تسک‌ها برای یکدیگر را می‌گیرد.

۳.۲ الگوهای طراحی برای ماشین‌های حالت

۱.۳.۲ الگوی **Single Event Receptor**

این الگو یک دریافت‌کننده رویداد را به کلاینت‌ها عرضه می‌کند که می‌تواند رویدادهای سنکرون و آسنکرون را دریافت کند. در این الگو، ورودی این دریافت‌کنند علاوه بر نوع رویدادی که رخ‌داده‌است، باید دارای داده‌های مربوط به رویداد نیز باشد.

۲.۳.۲ الگوی **Multiple Event Receptor**

در این الگو، برای هر یک از رویدادهای ممکن که توسط کلاینت رخ می‌دهد، یک دریافت‌کنند مجزا داریم. این الگو تنها برای رویدادهای سنکرون کاربرد دارد.

۳.۳.۲ الگوی **State Table**

این الگو یک نوع الگوی آفرینشی است که به صورت به خصوص برای ساخت ماشین حالت‌های با تعداد حالت‌های بسیار زیاد استفاده می‌شود. این الگو با ساخت یک جدول دوبعدی از نحوه گذار حالت‌ها از حالتی به حالت دیگر، ساختار ماشین حالت را می‌سازد. ساختار جدول به این شکل است که دارای تعدادی عملیات است که می‌گوید در صورت حضور در هر حالت و با آمدن هر رویدادی، باید چه عملیاتی انجام شود و حالت بعدی چیست.

۴.۳.۲ الگوی State

این الگو دقیقا همان الگوی State است که Gamma و دیگران در [۳] گفته‌اند. این الگو، با واسپاری حالت سیستم به یک شیء مجزا، وظیفه مدیریت حالت را به آن می‌دهد. در این الگو، تمامی رویدادهای دریافتی به این شیء پاس داده می‌شوند و او با توجه به این که حالت بعدی را می‌شناسد، خود را با شیء مربوط به حالت جدید جایگزین می‌کند.

۵.۳.۲ And States

۶.۳.۲ الگوی Decomposed And State

۴.۲ الگوهای امنیت و قابلیت اطمینان

۱.۴.۲ الگوی One's Complement

این الگو برای تشخیص آلودگی در حافظه است که ممکن است به دلیل اثرات بیرونی رخ داده‌باشد یا خطای سخت‌افزار باشد. با استفاده از این الگو می‌توان آلودگی را برای یک یا چند بیت از حافظه تشخیص داد. عملکرد کلی الگو به این شکل است که داده‌ها را دو بار ذخیره می‌کند. یک بار به صورت معمولی و یک بار به صورت 1's Complement. در زمان خواندن داده‌ها، اگر مقدار داده با 1's Complement گرفته‌شده آن دقیقا قرینه بودند، آن‌گاه داده بدون خطا ذخیره شده‌است و اگر اینگونه نباشد، نوشتن این داده با خطا مواجه شده‌بود.

۲.۴.۲ الگوی CRC

این الگو با استفاده از یک کد باینری با طول ثابت CRC، یک الگوریتم خطایابی ارائه می‌دهد که برای ساختار داده‌های بزرگ بسیار کاربرد خواهد داشت.

۳.۴.۲ الگوی Smart Data

این الگو با تولید گاردهایی روی داده‌ها و تعریف پیش‌شرط‌هایی روی آن‌ها در توابع مختلف تلاش می‌کند تا حد ممکن رفتار برنامه و توابع را به یک صورت Safe ایجاد کند. این شروط در زمان اجرا برنامه چک می‌شوند و نه در زمان کامپایل.

۴.۴.۲ الگوی Channel

کانال‌ها المان‌های نرم‌افزاری هستند که یک پردازش end-to-end انجام می‌دهند. این الگو با تعریف کانال‌هایی که مقدار زیادی اضافات در خود دارند، باعث می‌شوند که بتوانیم از آن‌ها در کاربردهای امنیت و قابلیت اطمینان استفاده کنیم. به طور مثال در این الگو، می‌توانیم با اینکه یک خطا تشخیص داده شده‌است، همچنان سرویس مورد نظر را ارائه دهیم.

۵.۴.۲ الگوی Protected Single Channel

این الگو بر پایه الگوی Channel ایجاد شده است. این الگو با پیاده‌سازی تعدای چک و گاردهای تعبیه‌شده در بخش‌های مهم کانال، می‌خواهد مراتبی برای ایجاد امنیت بسازد. میزان اضافات داده در این الگو کمتر از الگوی Channel است و به همین دلیل در صورت تشخیص خطا، نمی‌تواند به کار خود ادامه دهد اما در صورتی که خطا به صورت گذرا باشد، ممکن است که بتواند این کار را انجام دهد.

۶.۴.۲ الگوی Dual Channel

این الگو با ایجاد چند کانال و ایجاد اضافات در سطحی بالاتر، امنیت را تحقق می‌بخشد. اگر کانال‌ها از یک نوع باشند، این الگو Homo-geneous Redundancy Channel و در غیر این صورت Heterogeneous Redundancy Channel خوانده می‌شود. این الگو با تولید تعداد اضافه‌ای از کانال‌ها و مدیریت این که کدام یک از آن‌ها اکنون فعال هستند کار می‌کند. به طور کلی اگر در یکی از کانال‌ها خطایی رخ دهد، این الگو با سوییچ‌کردن روی یک کانال دیگر، سیستم را از حالت خطا خارج می‌کند.

۵.۲ الگوهای معماری زیربخش‌ها و اجزا

الگوهای معماری الگوهایی هستند که سطوح مختلف یک سیستم و نحوه چینش آن‌ها کنار یکدیگر را بیان می‌کنند. این الگوها ساختار زیربخش‌ها و اجزای درشت‌دانه یک سیستم هستند. این بخش براساس [۴] نوشته شده‌است.

۱.۵.۲ الگوی Layered

الگوی لایه‌ای دامنه‌های سیستم را بر اساس سطوح انتزاعی مختلف به صورت سلسله‌مراتبی سازمان‌دهی می‌کند. مفاهیم انتزاعی‌تر در یک دامنه با استفاده از مفاهیم ملموس‌تر در دامنه‌های دیگر پیاده‌سازی می‌شوند. این ساختار به متخصصان اجازه می‌دهد تا به طور موثر در زمینه تخصصی خود کار کنند بدون اینکه نیاز به درک تمامی جزئیات زیرین داشته باشند. به همین ترتیب، در توسعه نرم‌افزار، دامنه‌های انتزاعی با استفاده از دامنه‌های ملموس‌تر پیاده‌سازی می‌شوند که این امر موجب سازمان‌دهی و قابلیت تطبیق‌پذیری بیشتر بین پلتفرم‌های مختلف می‌شود.

۲.۵.۲ الگوی Five Layer

الگوی معماری پنج‌لایه یک تطبیق خاص از الگوی Layered است که برای ساختاردهی بسیاری از سیستم‌های نهفته و بی‌درنگ مفید است. این الگو معماری منطقی را به پنج لایه تقسیم می‌کند که این امر به توسعه‌دهندگان کمک می‌کند تا به راحتی ساختار سیستم‌های جدید را درک کنند. این الگو از قابلیت انتقال بین پلتفرم‌های مختلف پشتیبانی می‌کند و یک پلتفرم انتزاعی فراهم می‌کند که تطبیق برنامه‌ها را آسان‌تر می‌سازد. در حالی که این الگو بسیاری از مزایای الگوی Layered را دارد، از جمله کارایی بالا به دلیل تعداد کم لایه‌ها، ممکن است برای تجزیه کافی سیستم‌های پیچیده مناسب نباشد.

۳.۵.۲ الگوی Microkernel

الگوی معماری میکروکرنل برای سیستم‌هایی که دارای مجموعه‌ای اصلی از خدمات هستند که می‌توانند در زمان ساخت با خدمات اضافی گسترش یابند، مفید است. این الگو با ارائه قابلیت پیکربندی در زمان ساخت، قابلیت استفاده مجدد و تنظیم‌پذیری را افزایش می‌دهد و به توسعه‌دهندگان اجازه می‌دهد تا خدمات مورد نیاز برای یک برنامه را انتخاب کنند. یک مثال معمولی، سیستم‌عامل بی‌درنگ است که دارای خدمات اصلی مانند مدیریت وظایف و تخصیص حافظه است و می‌تواند با مؤلفه‌های ارتباطات، خدمات فایل، شبکه و میان‌افزار گسترش یابد. این الگو از مقیاس‌پذیری و تطبیق‌پذیری در طیف گسترده‌ای از برنامه‌ها، از سیستم‌های کوچک با محدودیت حافظه تا سیستم‌های پیچیده و شبکه‌ای، پشتیبانی می‌کند.

۴.۵.۲ الگوی Channel

الگوی معماری کانال در دو موقعیت اصلی مفید است: زمانی که داده‌ها در یک جریان داده به صورت ترتیبی از طریق چندین مرحله تبدیل می‌شوند و زمانی که اطمینان از قابلیت اطمینان بالا و ایمنی در برنامه‌های حیاتی مورد نیاز است. این الگو یک کانال را به عنوان یک لوله در نظر می‌گیرد که داده‌ها را به صورت ترتیبی پردازش می‌کند و هر عنصر داخلی یک عملیات ساده انجام می‌دهد. چندین کانال می‌توانند با پردازش همزمان عناصر مختلف داده‌ها توان عملیاتی را افزایش دهند و از طریق افزونگی قابلیت اطمینان و ایمنی را بهبود بخشند. این الگو به ویژه برای الگوریتم‌هایی که نیاز به تبدیل‌های مکرر دارند موثر است و امکان پردازش موازی کارآمد و تحمل خطا را فراهم می‌کند.

۵.۵.۲ الگوی Recursive Containment

الگوی تجزیه و تحلیل بازگشتی برای مدیریت سیستم‌های بسیار پیچیده با نیازمندی‌های فراوان مؤثر است. این الگو شامل شکستن سیستم به اجزای مرتبط در سطوح مختلف جزئیات است، مانند استفاده از میکروسکوپ با سطوح مختلف بزرگ‌نمایی. در هر سطح، اشیاء واسطهایی برای همتایان خود فراهم می‌کنند و وظایف را به اجزای کوچک‌تر داخلی تفویض می‌کنند، این تجزیه و تحلیل به صورت بازگشتی ادامه می‌یابد تا هر بخش دارای مسئولیت ساده و متمرکز شود. این رویکرد امکان تجزیه و تحلیل مقیاس‌پذیر را فراهم می‌کند و تأیید موارد استفاده بزرگ انتزاعی را در هر مرحله ممکن می‌سازد و سطوح مختلفی از جزئیات رفتار سیستم را ارائه می‌دهد.

۶.۵.۲ الگوی Hierarchical Control

الگوی کنترل سلسله‌مراتبی یک نسخه تخصصی از الگوی Recursive Containment است که الگوریتم‌های پیچیده کنترلی را بین اجزای مختلف توزیع می‌کند. این الگو از دو نوع واسط استفاده می‌کند: واسطهای کنترلی که نحوه دستیابی به رفتارها را نظارت و کنترل می‌کنند و واسطهای عملکردی که خدمات کنترل‌شده توسط واسطهای دیگر را فراهم می‌کنند. واسطهای کنترلی کیفیت خدمات، مانند دقت و صحت، را تعیین می‌کنند و سیاست‌های اجرایی را تنظیم می‌کنند. واسطهای عملکردی رفتار مطلوب را با استفاده از کیفیت خدمات و سیاست‌های تنظیم شده توسط واسط کنترلی اجرا می‌کنند. این الگو با استفاده از نمودارهای حالت برای هماهنگی اجزای زیرمجموعه و تجمیع اجزای جزء به کنترل‌کننده از طریق ترکیب، ساختار سلسله‌مراتبی قابل تنظیم و مقیاس‌پذیری را فراهم می‌کند. در این الگو، کنترل‌کننده وظیفه هماهنگی درخواست‌های خدمات به عناصر جزء را دارد و اغلب از نمودارهای حالت برای نشان دادن حالت‌های تنظیمات

اجزای زیرمجموعه استفاده می‌کند. این روش به ویژه زمانی مفید است که حالت‌های مختلف اجزای زیرمجموعه مستقل نباشند و با استفاده از نمودارهای حالت و انطباق حالت‌ها، سازگاری میان اجزا حفظ شود.

۷.۵.۲ الگوی Virtual Machine

الگوی ماشین مجازی اولویت را به قابلیت انتقال برنامه‌ها می‌دهد تا به کارایی در زمان اجرا، و برای برنامه‌هایی که نیاز به اجرای روی پلتفرم‌های مختلف دارند اما عملکرد حداکثری ضروری نیست، مناسب است. برنامه‌ها برای یک ماشین انتزاعی نوشته می‌شوند و یک ماشین مجازی نرم‌افزاری این دستورات را بر روی سخت‌افزار واقعی تفسیر می‌کند. این الگو انتقال برنامه‌ها به محیط‌های جدید را ساده می‌کند، زیرا فقط نیاز است ماشین مجازی برای پلتفرم هدف تطبیق داده شود. اگرچه برنامه‌ها ممکن است کندتر از برنامه‌های کامپایل شده بومی اجرا شوند، اما مزایای آن شامل ساده‌سازی انتقال و اندازه کوچکتر برنامه‌ها به دلیل اشتراک کتابخانه‌ها در داخل ماشین مجازی است. با این حال، ماشین‌های مجازی می‌توانند منابع زیادی مصرف کنند و ممکن است برای دستگاه‌های با محدودیت حافظه مناسب نباشند. در چنین شرایطی ممکن است الگویی مانند الگوی Microkernel مناسب‌تر باشد.

۸.۵.۲ معماری Component-Based

در UML، یک Component یک اثر زمان اجرا و یک واحد قابل جایگزینی اساسی در نرم‌افزار است که مشابه یک شیء بزرگ‌مقیاس شامل اشیاء کوچکتری است که واسط آن را پیاده‌سازی می‌کنند. Componentها دارای کپسوله‌سازی قوی و واسط‌های مستقل از زبان برنامه‌نویسی و کاملاً تعریف شده هستند. سیستم‌های مبتنی بر Component که از این اشیاء بزرگ‌مقیاس به عنوان واحدهای معماری استفاده می‌کنند، از نگهداری آسان، جداسازی عیوب، استقلال از زبان منبع، سادگی توسعه و قابلیت استفاده مجدد بهره‌مند می‌شوند. Componentها معمولاً اشیاء کوچکتری را برای هدف رفتاری مشترک زمان اجرا جمع می‌کنند. آنها دارای واسط‌های مبهم هستند، به این معنا که جزئیات داخلی آنها از کلاینت مخفی است که این امر جایگزینی را تضمین می‌کند اما ممکن است منجر به ناکارآمدی شود. الگوی معماری مبتنی بر مؤلفه معماری سیستم را قوی و قابل استفاده مجدد فراهم می‌کند اما ممکن است به دلیل استفاده از کل Componentها حتی اگر فقط بخشی از عملکرد آنها استفاده شود، منابع اضافی مصرف کند.

۹.۵.۲ الگوی ROOM

ROOM[۳] یک روش قدیمی‌تر است که پیش از UML وجود داشته است، با این حال UML می‌تواند ROOM را مدل‌سازی کند، همان‌طور که توسط تطابق UML-RT نشان داده شده است. ROOM نقش‌های خاصی برای رابط‌های دوطرفه به نام پورت‌ها شناسایی می‌کند و از کلاس‌های پروتکل برای کنترل این تعاملات استفاده می‌کند و کپسوله‌سازی قوی ارائه می‌دهد. این روش از نمودارهای حالت برای اجرای رفتار استفاده می‌کند و برای سیستم‌هایی با تعاملات پیچیده بین اشیاء درشت‌دانه مناسب است، چه توزیع شده باشند یا نه. این روش‌شناسی کپسول‌ها را معرفی می‌کند که می‌توانند زیرکپسول‌ها را شامل شوند و از پورت‌های رله برای ارسال پیام استفاده کنند. با وجود مزایای آن در مدیریت رابط‌ها و تعاملات پیچیده، ماهیت سنگین ROOM می‌تواند روابط ساده را پیچیده کند و باید با دقت انتخابی اعمال شود تا از محدودیت بیش از حد جلوگیری شود.

۶.۲ الگوهای معماری هم‌زمانی

همانطور که Douglass در [۴] می‌گوید، یکی از مسائل مهم در معماری سیستم‌ها، کنترل و زمان‌بندی بخش‌های مختلف معماری سیستم است. در UML، همزمانی از طریق Threadها مدیریت می‌شود که هر Thread در یک شیء «فعال» ریشه دارد و اشیاء «غیرفعال» را مدیریت می‌کند. توسعه‌دهندگان، Threadها را شناسایی کرده و هر کدام را به یک شیء «فعال» اختصاص می‌دهند. این اشیاء فعال مدیریت پیام و اجرا را در Threadهای خود انجام می‌دهند. این بخش نیز براساس [۴] نوشته شده‌است.

۱.۶.۲ الگوی Message Queuing

الگوی صف پیام روشی ساده برای ارتباط بین Thread ها فراهم می‌کند. علی‌رغم اینکه این روش نسبتاً سنگین برای اشتراک‌گذاری اطلاعات است، اما به‌طور گسترده استفاده می‌شود زیرا توسط بیشتر سیستم‌عامل‌ها پشتیبانی می‌شود و به راحتی قابل اثبات صحت است. این الگو از مشکلات Mutal Exclusion جلوگیری می‌کند زیرا هیچ منبع اشتراکی نیاز به محافظت ندارد، که همگام‌سازی را ساده کرده و یکپارچگی داده‌ها را تضمین می‌کند. در این الگو، اطلاعات به جای ارجاع، به صورت مقدار عبور می‌کنند و از مسائل فساد داده‌ای که در سیستم‌های همزمان رایج است، جلوگیری می‌کند. با این حال، این روش برای پردازش ساختارهای داده بزرگ کارایی کمتری دارد و اشتراک‌گذاری اطلاعات بسیار کارآمد را تسهیل نمی‌کند. (این الگو همان الگوی Queuing است که در [۱] گفته شده.)

[۳] Real-time Object Oriented Methodology

۲.۶.۲ الگوی Interrupt

وقفه‌ها به دلیل کارایی و اجرای سریع‌شان بسیار توصیه می‌شوند و در سیستم‌های بی‌درنگ و نهفته برای پاسخ به رویدادهای اضطراری ضروری هستند. این وقفه‌ها در مواقعی که پاسخ‌ها کوتاه و غیرقابل قطع هستند، عملکرد عالی دارند. با این حال، آنها برای همه موقعیت‌ها مناسب نیستند، به ویژه زمانی که پاسخ‌های طولانی‌تری مورد نیاز است یا سیستم بسیار فعال است. وقفه‌ها برای صف‌بندی پاسخ‌ها برای پردازش بعدی و به عنوان مکمل سایر استراتژی‌های همزمانی بهترین استفاده را دارند. باید دقت شود که پردازنده‌های وقفه کوتاه باشند تا از خرابی سیستم جلوگیری شود. اشتراک‌گذاری اطلاعات بین پردازنده‌های وقفه چالش‌برانگیز است زیرا نیاز به دسترسی محافظت‌شده بدون بلاک کردن دارند. (این الگو همان الگوی Interrupt است که در [۱] گفته‌شده.)

۳.۶.۲ الگوی Guarded Call

الگوی فراخوانی محافظت‌شده راهی برای دستیابی به همگام‌سازی به موقع بین Threadها از طریق فراخوانی همزمان متدها در یک Thread دیگر ارائه می‌دهد و از Mutal Exclusion Semaphores برای جلوگیری از فساد داده و Deadlock استفاده می‌کند. در حالی که ارتباطات غیرهمزمان مانند الگوی Message Queuing اغلب منجر به تبادل اطلاعات کندتر می‌شود، الگوی فراخوانی محافظت‌شده با اجازه دادن به فراخوانی مستقیم متدها، زمان پاسخ‌دهی سریع‌تری را تضمین می‌کند. این الگو زمانی که همگام‌سازی فوری مورد نیاز است بسیار مفید است، اگرچه باید با دقت پیاده‌سازی شود تا از مشکلات Mutal Exclusion جلوگیری شود. اگر منابع قفل باشند، بدون تحلیل مناسب نمی‌توان پاسخ‌دهی به موقع را تضمین کرد. (این الگو همان الگوی Guarded Call است که در [۱] گفته‌شده.)

۴.۶.۲ الگوی Rendezvous

الگوی Rendezvous نسخه ساده‌تری از الگوی Guarded Call است که برای همگام‌سازی Threadها یا اجازه اشتراک‌گذاری داده‌ها بین آنها استفاده می‌شود. این الگو از یک شیء Rendezvous برای مدیریت همگام‌سازی استفاده می‌کند که ممکن است شامل داده‌های اشتراکی یا فقط اعمال سیاست‌های همگام‌سازی باشد. ساده‌ترین شکل آن، الگوی Thread Barrier است که Threadها را بر اساس تعداد مشخصی که در یک نقطه ثبت‌نام می‌کنند، همگام می‌کند. پیش‌شرط‌ها برای همگام‌سازی باید برآورده شوند که اغلب توسط ماشین‌های حالت در زبان‌های طراحی مانند UML مدیریت می‌شوند. این الگو تضمین می‌کند که Threadها منتظر می‌مانند تا همه شرایط برآورده شود و سپس ادامه می‌دهند. این الگو بسیار انعطاف‌پذیر است، برای نیازهای همگام‌سازی پیچیده کاربرد دارد و به خوبی با تعداد زیادی از Threadها و شروط، مقیاس‌پذیر است. (این الگو همان الگوی Rendezvous است که در [۱] گفته‌شده.)

۵.۶.۲ الگوی Cyclic Execution

الگوی Cyclic Execution در سیستم‌های کوچک یا سیستم‌هایی که نیاز به اجرای قابل پیش‌بینی دارند، مانند سیستم‌های هوانوردی، به طور گسترده‌ای استفاده می‌شود. این الگو با سادگی و پیش‌بینی‌پذیری خود، پیاده‌سازی آسانی دارد و برای محیط‌های محدود به حافظه که یک سیستم‌عامل بی‌درنگ کامل عملی نیست، مناسب است. این الگو وظایف را در یک حلقه ثابت و تکراری اجرا می‌کند و اطمینان می‌دهد که هر وظیفه به نوبت اجرا می‌شود. سادگی آن نقطه قوت اصلی آن است، اما انعطاف‌پذیری ندارد و برای رسیدگی به رویدادهای با ضرب‌الاجل‌های محدود بهینه نیست. وظایف نمی‌توانند در زمان اجرا اضافه یا حذف شوند و سیستم به تنظیمات زمانی حساس است. وظایف نادرست می‌توانند کل سیستم را مختل کنند و الگو در شرایط بار زیاد ناپایدار است. با وجود محدودیت‌ها، برای سیستم‌های کوچک و پایدار با دینامیک‌های قابل درک مناسب است. (این الگو همان الگوی Cyclic Executive است که در [۱] گفته‌شده.)

۶.۶.۲ الگوی Round Robin

الگوی Round Robin با دادن فرصت به همه وظایف برای پیشرفت، اطمینان از عدالت در اجرای وظایف را فراهم می‌کند و برای سیستم‌هایی مناسب است که پیشرفت کلی سیستم مهم‌تر از برآورده شدن ضرب‌الاجل‌های خاص است. برخلاف الگوی Cyclic Execution، الگوی Round Robin از پیش‌دستی زمانی استفاده می‌کند و مانع از متوقف شدن سیستم توسط یک وظیفه نادرست می‌شود. با این حال، این الگو محدودیت‌هایی مانند پاسخ‌دهی نامطلوب به رویدادها و عدم پیش‌بینی‌پذیری در شرایط بار بسیار زیاد را دارد. با این که این الگو در مقایسه با الگوی Cyclic Execution بهتر به تعداد بیشتری از وظایف مقیاس‌پذیر است، اما با افزایش تعداد وظایف، می‌تواند منجر به تشدید وظایف شود و زمان مؤثر هر وظیفه کاهش یابد. مکانیزم‌های اشتراک‌گذاری داده‌ها ابتدایی هستند و پیاده‌سازی مدل‌های پیچیده را دشوار می‌کنند.

۷.۶.۲ الگوی Static Priority

(این الگو همان الگوی Static Priority است که در [۱] گفته‌شده.)

۸.۶.۲ الگوی Dynamic Priority

الگوی Dynamic Priority، اولویت وظایف را بر اساس فوریت در زمان اجرا تنظیم می‌کند و معمولاً از استراتژی نزدیک‌ترین ضرب‌الاجل استفاده می‌کند، جایی که وظیفه‌ای که نزدیک‌ترین ضرب‌الاجل را دارد بالاترین اولویت را دریافت می‌کند. این روش بهینه است زیرا اگر وظایف بتوانند توسط هر الگوریتمی زمان‌بندی شوند، می‌توانند توسط این الگوریتم نیز زمان‌بندی شوند. با این حال، این الگو ناپایدار است، به این معنی که پیش‌بینی اینکه کدام وظایف در شرایط بار زیاد شکست می‌خورند، ممکن نیست. این الگو برای سیستم‌های پیچیده با وظایف تقریباً برابر از نظر اهمیت و جایی که تحلیل استاتیک غیرممکن است، مناسب است. در مقابل، الگوی Static Priority برای سیستم‌های ساده‌تر که وظایف و زمان‌بندی آن‌ها می‌تواند به دقت شناخته و برنامه‌ریزی شود، بهتر است.

۷.۲ الگوهای معماری حافظه

در این بخش به بررسی الگوهای مدیریت حافظه و به اشتراک‌گذاری منابع در سیستم نرم‌افزاری می‌پردازیم. این دسته از الگوها از [۴] آورده شده‌است.

۱.۷.۲ الگوی Static Allocation

الگوی Static Allocation برای سیستم‌های ساده با بارهای حافظه قابل پیش‌بینی و ثابت طراحی شده است. این الگو از تخصیص حافظه پویا اجتناب می‌کند تا مشکلاتی مانند زمان‌بندی غیرقابل‌پیش‌بینی و تکه‌تکه شدن حافظه را از بین ببرد. در عوض، همه اشیاء در زمان راه‌اندازی سیستم تخصیص داده می‌شوند که منجر به زمان راه‌اندازی طولانی‌تر اما عملکرد زمان اجرای قابل پیش‌بینی‌تر و سریع‌تر می‌شود. این الگو زمانی مفید است که نیازهای حافظه در بدترین حالت مشخص باشد و حافظه کافی برای برآورده کردن آن‌ها وجود داشته باشد. این الگو طراحی و نگهداری سیستم را ساده می‌کند اما انعطاف‌پذیری کمتری دارد و ممکن است به حافظه بیشتری نسبت به تخصیص پویا نیاز داشته باشد. مزیت اصلی آن حذف تکه‌تکه شدن حافظه و بهبود پیش‌بینی‌پذیری است.

۲.۷.۲ الگوی Pool Allocation

الگوی Pool Allocation برای سیستم‌هایی که بسیار پویا هستند و تخصیص ایستا برای آن‌ها مناسب نیست، اما همچنان می‌خواهند از مشکلات تخصیص حافظه پویا اجتناب کنند، مناسب است. این الگو شامل ایجاد Poolهایی از اشیاء در زمان راه‌اندازی است که بر اساس درخواست کلاینت‌ها در دسترس قرار می‌گیرند. این الگو برای سیستم‌هایی که به مجموعه‌ای از اشیاء برای اهداف مختلف در زمان‌های مختلف اجرای سیستم نیاز دارند، مانند اشیاء داده یا پیام، که نمی‌توان در زمان طراحی به‌طور بهینه پیش‌بینی یا توزیع کرد، مفید است. اشیاء از Poolها تخصیص داده می‌شوند، استفاده می‌شوند و سپس به آن بازگردانده می‌شوند، بنابراین از مشکلات تخصیص حافظه در زمان اجرا و تکه‌تکه شدن حافظه اجتناب می‌شود. با این حال، نیاز است که تعداد بهینه اشیاء مختلف در زمان طراحی تعیین شود و این الگو اجازه رشد دینامیک تقاضای سیستم را نمی‌دهد.

۳.۷.۲ الگوی Fixed Sized Buffer

این الگو مشکل تکه‌تکه شدن حافظه را در تخصیص حافظه پویا برطرف می‌کند، که این مسئله برای سیستم‌های بی‌درنگ نهفته که باید برای مدت‌های طولانی به‌طور قابل اعتماد عمل کنند، بسیار مهم است. این الگو با استفاده از بلوک‌های حافظه با اندازه ثابت از تکه‌تکه شدن جلوگیری می‌کند، اگرچه منجر به هدر رفتن بخشی از حافظه به دلیل استفاده غیر بهینه می‌شود. این مصالحه معمولاً در بسیاری از سیستم‌های عامل بی‌درنگ قابل قبول است که اغلب از تخصیص بلوک‌های با اندازه ثابت به صورت داخلی پشتیبانی می‌کنند.

۴.۷.۲ الگوی Smart Pointer

الگوی اشاره‌گر هوشمند یک راه‌حل طراحی برای کاهش مشکلات رایج مربوط به اشاره‌گرها در برنامه‌نویسی است، مانند نشت حافظه، Dangling Pointers، اشاره‌گرهای مقداردهی‌نشده، و نقص‌های محاسباتی اشاره‌گر. با کپسوله کردن اشاره‌گرها در اشیاء، اشاره‌گرهای هوشمند قوانین مدیریت صحیح حافظه را از طریق سازنده‌ها و مخرب‌ها اعمال می‌کنند و در نتیجه قابلیت اطمینان و نگهداری را افزایش می‌دهند. در حالی که مفید هستند، نیاز به انضباط شدید دارند تا از مخلوط کردن اشاره‌گرهای خام و هوشمند جلوگیری کنند و در زمینه‌های Multi-Thread با دقت مدیریت شوند. علاوه بر این، آنها هنوز هم می‌توانند منجر به نشت حافظه شوند اگر ارجاعات چرخشی بین اشیاء وجود داشته باشد.

۵.۷.۲ الگوی Garbage Collection

الگوی Garbage Collection به مشکلات مدیریت حافظه مانند نشت حافظه و Dangling Pointer با خودکارسازی فرآیند آزادسازی حافظه پرداخته و نیاز به آزادسازی صریح حافظه توسط برنامه‌نویسان را حذف می‌کند. این الگو به طور قابل‌توجهی مشکلات مرتبط با حافظه را کاهش داده و برای سیستم‌های با دسترسی بالا که نیاز به اجرای طولانی‌مدت بدون راه‌اندازی مجدد دارند، ایده‌آل است. با این حال، این الگو به دلیل طبیعت دوره‌ای خود، سربار زمان اجرا و عدم پیش‌بینی‌پذیری در اجرا را به همراه دارد و مشکل تکه‌تکه شدن حافظه را حل نمی‌کند که می‌توان آن را با استفاده از الگوی Garbage Compactor مدیریت کرد.

۶.۷.۲ الگوی Garbage Compactor

الگوی فشرده‌سازی زباله نسخه‌ای از الگوی Garbage Collection است که به تکه‌تکه شدن حافظه نیز می‌پردازد. این الگو با نگه‌داری دو بخش حافظه در Heap و جابجایی دوره‌ای اشیای زنده از یک بخش به بخش دیگر، حافظه آزاد را به صورت پیوسته نگه می‌دارد. این الگو تکه‌تکه شدن حافظه را حذف کرده و بلوک‌های پیوسته حافظه آزاد را فراهم می‌کند و اطمینان می‌دهد که درخواست‌های تخصیص حافظه همیشه در صورت وجود حافظه کافی، برآورده می‌شوند. با این حال، به دلایل گفته‌شده، این الگو به دو برابر حافظه بیشتر نسبت به الگوی Garbage Collection نیاز دارد و به دلیل نیاز به کپی کردن اشیا بین بخش‌ها، بار اضافی بر CPU وارد می‌کند. این الگو برای سیستم‌هایی با محدودیت‌های سخت‌گیرانه در حافظه مناسب نیست و نیاز به مدیریت دقیق نحوه finalizeکردن اشیا دارد.

۸.۲ الگوهای معماری منابع

این دسته از الگوها که در [۴] نام برده شده‌اند، به ما می‌گویند که چگونه می‌توان منابع را در یک سیستم بی‌درنگ نهفته به اشتراک گذاشت و مدیریت کرد. در سیستم‌های نهفته، منابع بسیار محدود هستند و این الگوها از این منظر بسیار حائز اهمیت هستند.

۱.۸.۲ الگوی Critical Section

(این الگو همان الگوی Critical Region است که در اینجا با نامی دیگر بیان شده‌است.)

۲.۸.۲ الگوی Priority Inheritance

الگوی Priority Inheritance برای کاهش وارونگی اولویت در سیستم‌های بی‌درنگ نهفته طراحی شده است که با تنظیم اولویت‌های وظایفی که منابع را قفل می‌کنند، انجام می‌شود. این الگو، اگرچه کامل نیست، اما وارونگی اولویت را با سربار اجرایی نسبتاً کم به حداقل می‌رساند. وارونگی اولویت می‌تواند به خرابی‌های سیستم منجر شود که تشخیص آنها دشوار است، زیرا وظایف گاهی مهلت‌ها را از دست می‌دهند بدون اینکه علل واضحی داشته باشند. این الگو تضمین می‌کند که یک وظیفه با اولویت بالا، در بدترین حالت، تنها توسط یک وظیفه با اولویت پایین‌تر مسدود می‌شود و به این ترتیب وارونگی اولویت بی‌نهایت را حل می‌کند.

هنگامی که چندین منبع قفل شده باشند، مسدودسازی زنجیره‌ای رخ می‌دهد، به طوری که یک وظیفه دیگری را در یک زنجیره مسدود می‌کند. با این حال، این الگو به طور قابل توجهی مسدودسازی بی‌نهایت را محدود می‌کند، به طوری که تعداد وظایف مسدود شده در هر زمان کمتر از تعداد وظایف و منابع قفل شده است. سربار زمانی از مدیریت اولویت‌های وظایف در هنگام مسدودسازی و رفع مسدودسازی ناشی می‌شود، اما اگر مسدودسازی نادر باشد، سربار کمی باقی می‌ماند.

این الگو مشکلات Deadlock را حل نمی‌کند و می‌تواند باعث ایجاد سربار ناشی از جابه‌جایی وظایف و تنظیمات اولویت شود، به ویژه اگر رقابت بر سر منابع بالا باشد.

۳.۸.۲ الگوی Highest Locker

این الگو به هدف کاهش وارونگی اولویت، سقف اولویتی برای هر منبع تعیین می‌کند. وظیفه‌ای که مالک منبع است، در بالاترین سقف اولویت از بین همه منابعی که در اختیار دارد اجرا می‌شود، به شرطی که وظایف با اولویت بالاتر را مسدود کند. این الگو که نوعی تصحیح شده از الگوی Priority Inheritance است، وارونگی اولویت را به یک سطح واحد محدود می‌کند، به شرطی که وظایف در حین داشتن منابع خود را معلق نکنند. بر خلاف الگوی Priority Inheritance، این الگو مانع از مسدودسازی زنجیره‌ای در صورت پیش‌دستی وظیفه‌ای در حین مالکیت منبع می‌شود.

این الگو سقف اولویت‌ها را در زمان طراحی با شناسایی بالاترین اولویت بین کلاینت‌های هر منبع و افزودن یک واحد به آن تعریف می‌کند. این الگو به طور موثری وارونگی اولویت را محدود می‌کند، اما می‌تواند منجر به بلوکه شدن بیشتر در این سطح واحد نسبت به روش‌های دیگر شود. به عنوان مثال، اگر یک وظیفه با اولویت پایین یک منبع با سقف اولویت بالا را قفل کند، وظایف با اولویت متوسط ممکن

است بیشتر مسدود شوند. برای مدیریت این وضعیت، می‌توان افزایش اولویت را تا زمانی که وظیفه دیگری تلاش برای قفل کردن منبع کند، به تعویق انداخت.

این الگو از Deadlock جلوگیری می‌کند، به شرطی که تسک‌ها در حین داشتن منابع خود را معلق نکنند، زیرا اولویت تسک قفل‌کننده بالاتر از دیگر مشتریان منبع است. با این حال، سربار محاسباتی در مدیریت سقف اولویت‌ها و اطمینان از اجرای صحیح تسک‌ها بدون تعلیق وجود دارد.

۴.۸.۲ الگوی **Priority Ceiling**

این الگو یک روش پیچیده برای حل مشکلات وارونگی اولویت و Deadlock در سیستم‌های چندوظیفه‌ای بسیار قابل اعتماد است. این الگو وارونگی اولویت و زمان‌های مسدود شدن وظایف را محدود می‌کند و از وقوع Deadlockهای ناشی از رقابت بر سر منابع جلوگیری می‌کند. اگرچه پیچیده‌تر و با سربار بیشتری نسبت به روش‌های دیگر مانند الگوی Highest Locker است، این الگو تضمین می‌کند که یک وظیفه با اولویت بالا تنها می‌تواند توسط یک وظیفه با اولویت پایین‌تر که مالک یک منبع مورد نیاز است مسدود شود.

در این الگو، ممکن است یک وظیفه در حال اجرا نتواند به یک منبع دسترسی پیدا کند حتی اگر آن منبع قفل نباشد، اگر سقف اولویت آن منبع کمتر از سقف منابع سیستم فعلی باشد. این امر با حذف احتمال وقوع شرایط انتظار حلقوی به جلوگیری از Deadlock کمک می‌کند. با این حال، پیچیدگی و سربار محاسباتی افزوده باعث می‌شود این الگو کمتر توسط سیستم‌عامل‌های بی‌درنگ تجاری پشتیبانی شود و اغلب نیاز به افزونه‌های سفارشی برای پیاده‌سازی دارد.

۵.۸.۲ الگوی **Simultaneous Locking**

(این الگو همان الگوی Simultaneous Locking است که در [۱] بیان شده‌است.)

۶.۸.۲ الگوی **Ordered Locking**

(این الگو همان الگوی Ordered Locking است که در [۱] بیان شده‌است.)

۹.۲ الگوهای معماری توزیع
۱.۹.۲ الگوی **Shared Memory**
۲.۹.۲ الگوی **Remote Method Call**
۳.۹.۲ الگوی **Observer**
۴.۹.۲ الگوی **Data Bus**
۵.۹.۲ الگوی **Proxy**
۶.۹.۲ الگوی **Broker**
۱۰.۲ الگوهای معماری امنیت و قابلیت اطمینان
۱.۱۰.۲ الگوی **Protected Single Channel**
۲.۱۰.۲ الگوی **Homogeneous Redundancy**
۳.۱۰.۲ الگوی **Triple Modular Redundancy**
۴.۱۰.۲ الگوی **Heterogeneous Redundancy**
۵.۱۰.۲ الگوی **Monitor-Actuator**
۶.۱۰.۲ الگوی **Sanity Check**
۷.۱۰.۲ الگوی **Watchdog**
۸.۱۰.۲ الگوی **Safety Executive**
۱۱.۲ الگوهای سخت‌افزاری برای سیستم‌های **Safety-Critical**
۱.۱۱.۲ الگوی **Homogeneous Duplex**
۲.۱۱.۲ الگوی **Heterogeneous Duplex**
۳.۱۱.۲ الگوی **Triple Modular Redundancy**
۴.۱۱.۲ الگوی **M-Out-Of-N**
۵.۱۱.۲ الگوی **Monitor-Actuator**
۶.۱۱.۲ الگوی **Sanity Check**
۷.۱۱.۲ الگوی **Watchdog**
۸.۱۱.۲ الگوی **Safety Executive**
۱۲.۲ الگوهای نرم‌افزاری برای سیستم‌های **Safety-Critical**
۱.۱۲.۲ الگوی **N-Version Programming**
۲.۱۲.۲ الگوی **Recovery Block**
۳.۱۲.۲ الگوی **Acceptance Voting**
۴.۱۲.۲ الگوی **N-Self Checking Programming**
۵.۱۲.۲ الگوی **Recovery Block with Backup Voting**
۱۳.۲ الگوهای ترکیبی سخت‌افزار و نرم‌افزار برای سیستم‌های **Safety-Critical**
۱.۱۳.۲ الگوی **Protected Single Channel**
۲.۱۳.۲ الگوی **3-Level Safety Monitoring**

۳ تحلیل

۴ مراجع

[۱] Douglass, Bruce Powel. Design patterns for embedded systems in C: an embedded software engineering toolkit. Elsevier, 2010.

[۲] Zalewski, Janusz. "Real-time software architectures and design patterns: Fundamental concepts and their consequences." Annual Reviews in Control 25 (2001): 133-146.

[۳] Gamma, Erich, et al. "Design patterns: Abstraction and reuse of object-oriented design." ECOOP'93—Object-Oriented Programming: 7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7. Springer Berlin Heidelberg, 1993.

[۴] Douglass, Bruce Powel. Real-time design patterns: robust scalable architecture for real-time systems. Addison-Wesley Professional, 2003.

[۵] Armoush, Ashraf. Design patterns for safety-critical embedded systems. Diss. RWTH Aachen University, 2010.