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

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.

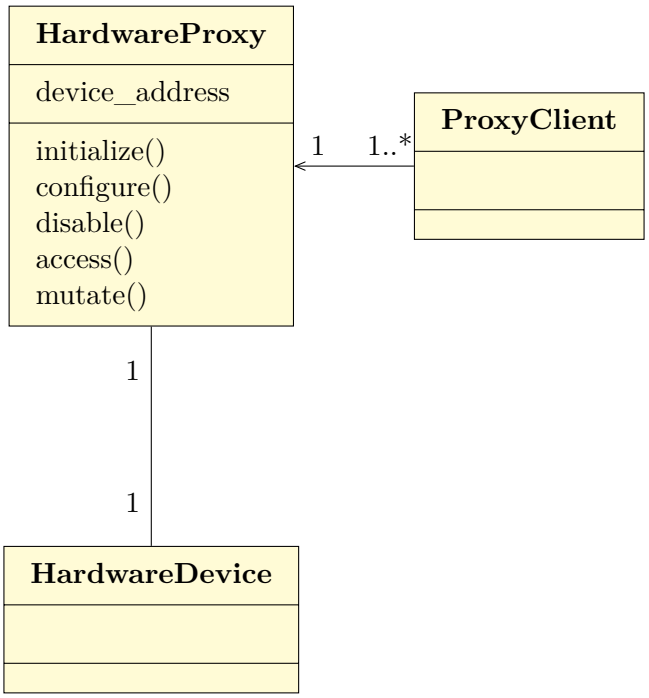
۲ پیشینه پژوهش

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 [۱] 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 کلاینت‌های 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:

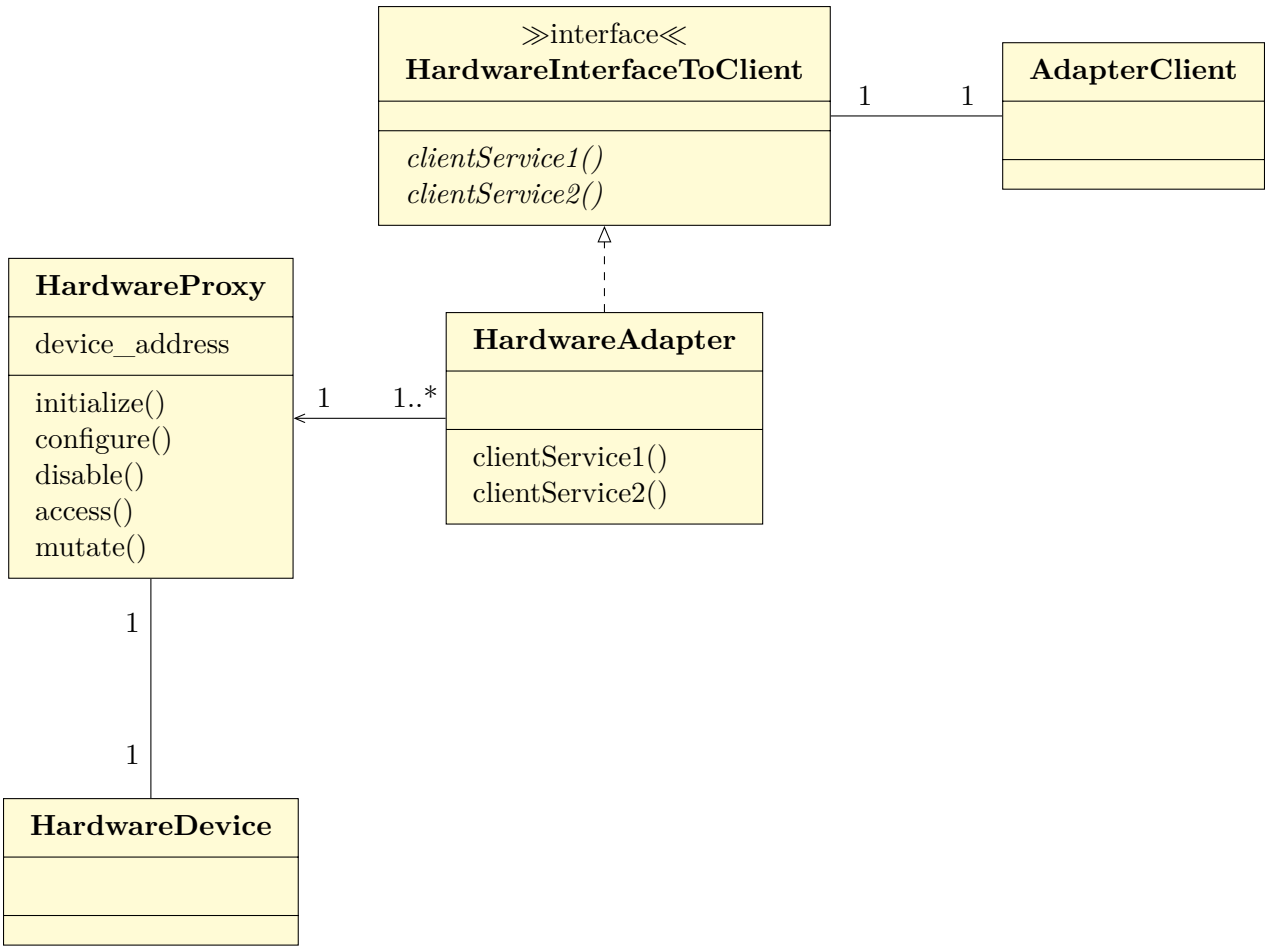
- initialize: این تابع برای آماده‌سازی اولیه ارتباط با سخت‌افزار استفاده می‌شود و معمولا تنها یک بار صدا زده می‌شود.
- configure: این تابع برای ارسال تنظیمات برای سخت‌افزار استفاده می‌شود. معمولا باید در سخت‌افزار تنظیماتی قرار داده‌شود که آن را قابل استفاده کند.
- disable: این تابع برای غیرفعال‌کردن 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.

این الگو بسیار رایج است و مزایای کپسوله‌سازی رابط سخت‌افزار و جزئیات کدگذاری را فراهم می‌کند، به طوری که تغییرات رابط سخت‌افزار بدون نیاز به تغییر در کلاینت‌ها انجام می‌شود. این کپسوله‌سازی می‌تواند تأثیر منفی بر عملکرد زمان اجرا داشته باشد، زیرا کاربران از فرمت اصلی داده‌ها آگاه نیستند. با این حال، آگاهی کلاینت‌ها از جزئیات کدگذاری باعث پیچیدگی در نگه‌داشت سیستم می‌شود.

۲.۱.۲ الگوی **Hardware Adapter**

این الگو [۱] 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.



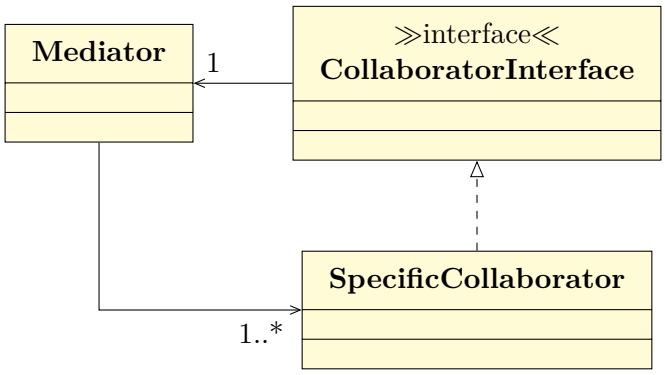
شکل ۲: دیاگرام کلاس Hardware Adapter

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.

استفاده از این الگو اجازه می‌دهد پروکسی‌های سخت‌افزار و دستگاه‌های مرتبط در برنامه‌های مختلف بدون تغییر استفاده شوند و برنامه‌های موجود نیز بدون تغییر از دستگاه‌های سخت‌افزاری مختلف استفاده کنند. Adapter به‌عنوان پل ارتباطی بین پروکسی سخت‌افزار و برنامه عمل می‌کند، که تغییر یا استفاده مجدد از دستگاه‌های سخت‌افزاری را آسان‌تر، سریع‌تر و کم‌خطاتر می‌سازد. هزینه استفاده از این الگو افزایش سطح انتزاع و کاهش اندک عملکرد زمان اجرا است.

۳.۱.۲ الگوی Mediator

این الگو [۱] 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.



شکل ۳: دیاگرام کلاس Mediator

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.

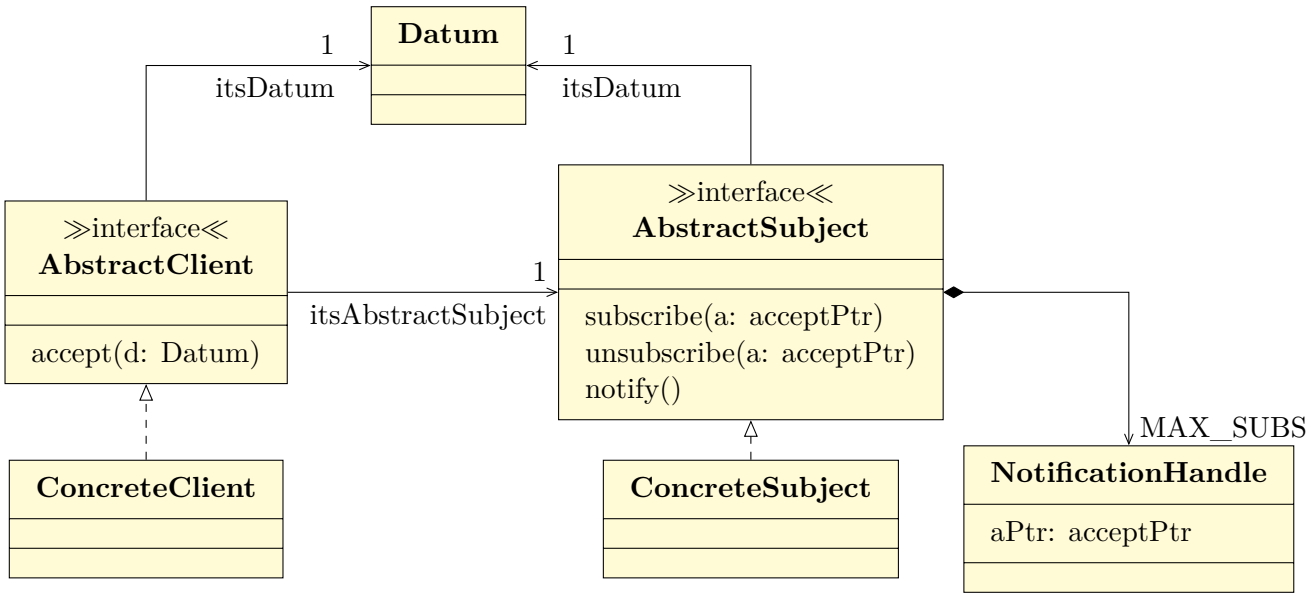
این الگو یک میانجی ایجاد می‌کند که هماهنگی بین مجموعه‌ای از عملگرهای همکار را بدون نیاز به اتصال مستقیم آن‌ها انجام می‌دهد، که طراحی کلی را ساده‌تر می‌کند. میانجی به جای تماس مستقیم همکاران با یکدیگر، اطلاعیه‌ها را دریافت و به صورت کلی پاسخ می‌دهد. با توجه به نیاز بسیاری از سیستم‌های نهفته به واکنش سریع، تأخیرها می‌توانند اثرات ناپایداری داشته باشند. بنابراین، واکنش به موقع میانجی در همکاری دوطرفه با عملگرها بسیار مهم است.

۴.۱.۲ الگوی Observer

یکی از پرکاربردترین الگوها در حوزه سیستم‌های نهفته، الگوی Observer [۱] 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.

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).

این الگو فرآیند توزیع داده‌ها به مجموعه‌ای از کلاینت‌ها را که ممکن است در زمان طراحی مشخص نباشند، ساده می‌کند و مدیریت داینامیک لیست کلاینت‌های علاقه‌مند در زمان اجرا را فراهم می‌کند. این الگو رابطه اصلی کلاینت-سرور را حفظ می‌کند و از طریق مکانیزم اشتراک‌گذاری، انعطاف‌پذیری در زمان اجرا ارائه می‌دهد. بهره‌وری محاسباتی حفظ می‌شود زیرا کلاینت‌ها تنها زمانی که مناسب است به‌روزرسانی می‌شوند؛ سیاست رایج این است که مشتریان در زمان تغییر داده‌ها به‌روزرسانی شوند، اما هر سیاست مناسب دیگری نیز می‌تواند اعمال شود.



شکل ۴: دیاگرام کلاس Observer

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 [۱] 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.

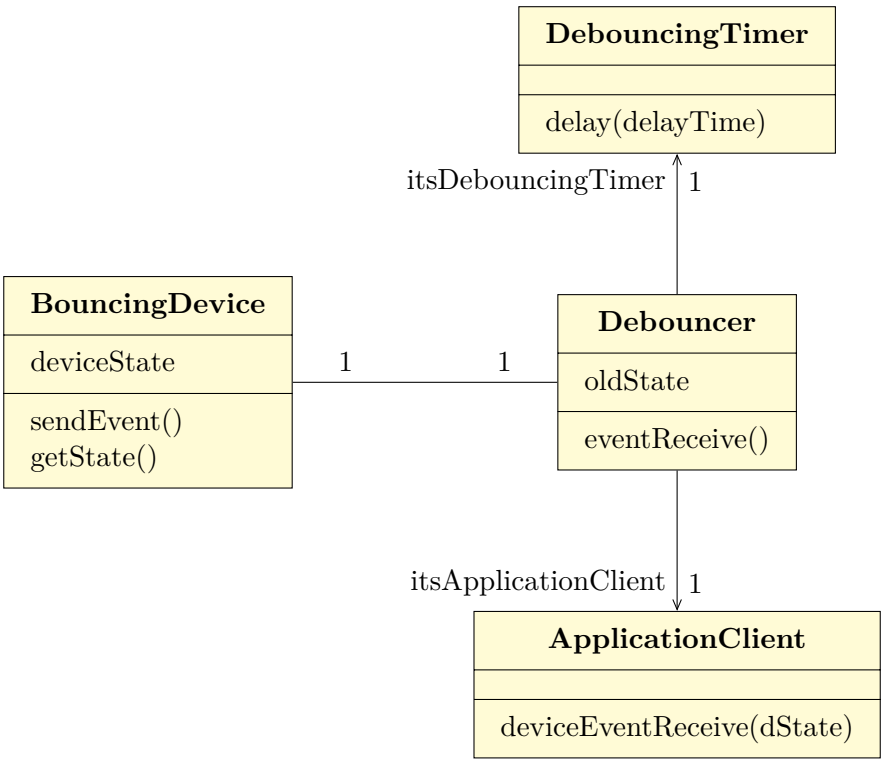
در این ساختار، BouncingDevice همان سخت‌افزار مورد بررسی است. تابع sendEvent می‌تواند یک نوع اینتراپت سخت‌افزاری باشد که نرم‌افزار را از تغییر در سخت‌افزار باخبر می‌سازد و getState می‌تواند یک عملیات خواندن از حافظه باشد. کلاس Debouncer وظیفه ارائه حالت پایدار سخت‌افزار به کلاینت را دارد. این کار با استفاده 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**

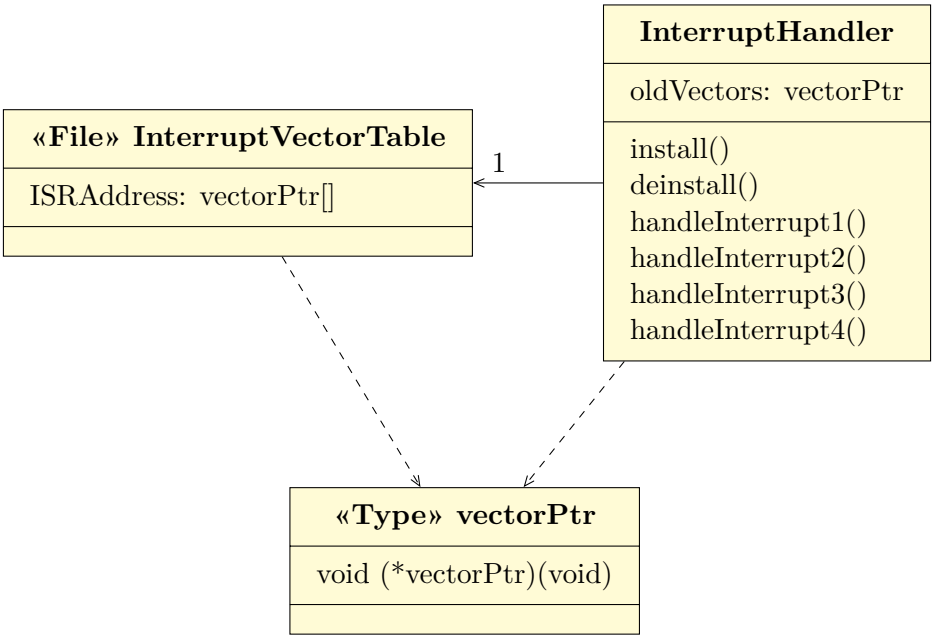
یکی از واحدهای مهم در سیستم‌های سخت‌افزاری، واحد Interrupt است. Interrupt برای هندل‌کردن وقایعی است که توسط سخت‌افزار جرقه زده می‌شوند. زمانی که یک interrupt رخ می‌دهد، نرم‌افزار فرایند کاری اصلی خود را متوقف کرده و یک فرایند رسیدگی به interrupt اتفاق‌افتاده آغاز می‌شود. با انجام این فرایند و رسیدگی به interrupt، فرایند اصلی نرم‌افزار دوباره از سر گرفته می‌شود. ساختار این الگو [۱] 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.

این الگو امکان پردازش سریع رویدادهای مهم را فراهم می‌کند. با وقفه در پردازش معمولی (به شرطی که interruptها غیرفعال نشده باشند) باید در زمان پردازش حساس به زمان با احتیاط استفاده شود. وقفه‌ها در حین اجرای Interrupt Service Routine (ISR) غیرفعال می‌شوند، بنابراین ISRها باید به سرعت اجرا شوند تا وقفه‌های دیگر از دست نروند. ISRها باید کوتاه باشند و در هنگام استفاده از آن‌ها برای فراخوانی خدمات دیگر سیستم باید دقت شود. برای به اشتراک گذاشتن داده‌ها، ISR ممکن است نیاز به صف‌بندی داده‌ها و بازگشت سریع داشته باشد تا برنامه در آینده داده‌ها را در صف پیدا کند. این روش زمانی مفید است که جمع‌آوری داده‌ها مهم‌تر از پردازش آن‌ها باشد. مشکلات زمانی رخ می‌دهند که پردازش ISR بیش از حد طولانی شود، یا اشتباهاتی در پیاده‌سازی وقفه‌ها را غیرفعال



شکل ۵: دیاگرام کلاس Debouncing

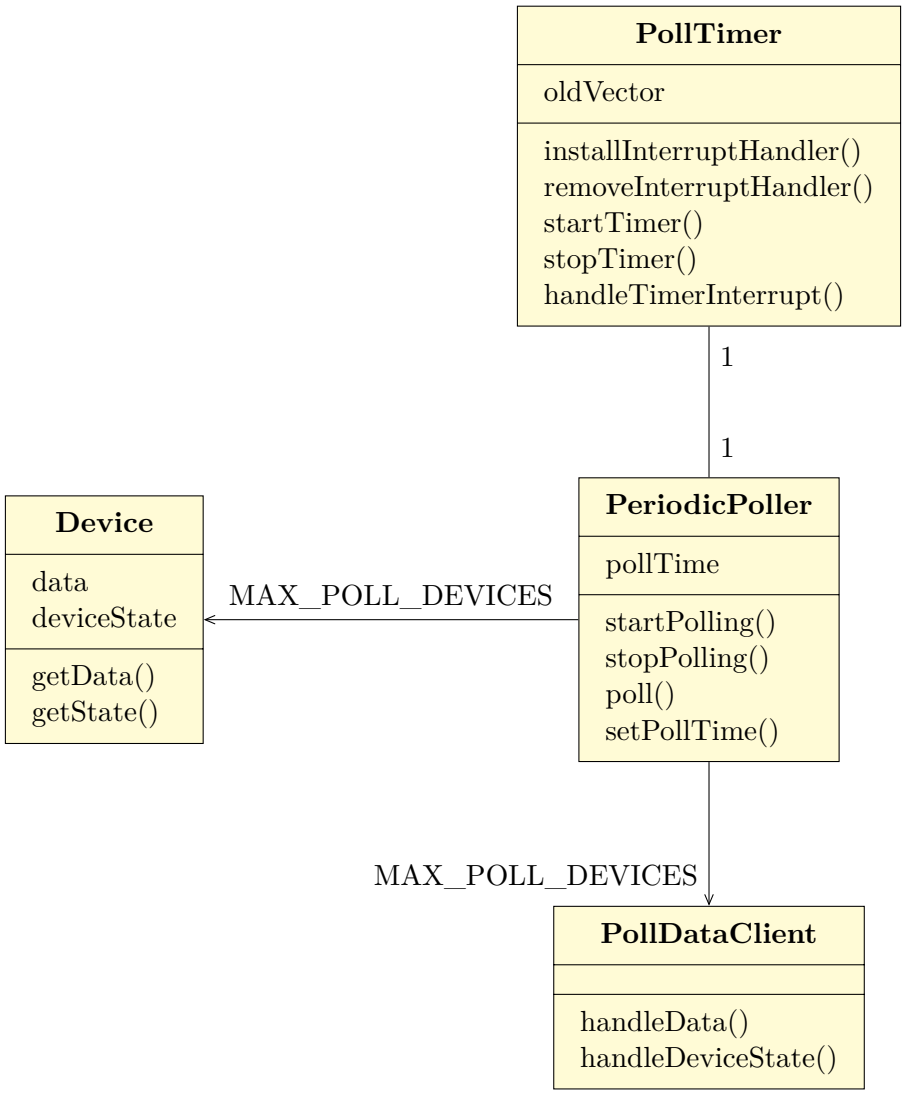


شکل ۶: دیاگرام کلاس Interrupt

کند، یا شرایط رقابتی یا بن‌بست در منابع مشترک رخ دهد. شرایط رقابتی زمانی رخ می‌دهد که نتیجه به ترتیب اجرای دستورات بستگی دارد، اما این ترتیب ناشناخته است. بن‌بست زمانی است که دو عنصر منتظر شرطی هستند که اصولاً اتفاق نمی‌افتد. این مشکلات با استفاده از قفل‌های متقارن قابل حل هستند، اما می‌توانند منجر به بن‌بست شوند اگر ISR منتظر قفل بماند. راه‌حل این است که ISR نباید منتظر قفل بماند و باید داده‌های جدید را رد کند یا از منابع مشترک دوگانه استفاده کند.

۷.۱.۲ الگوی Polling

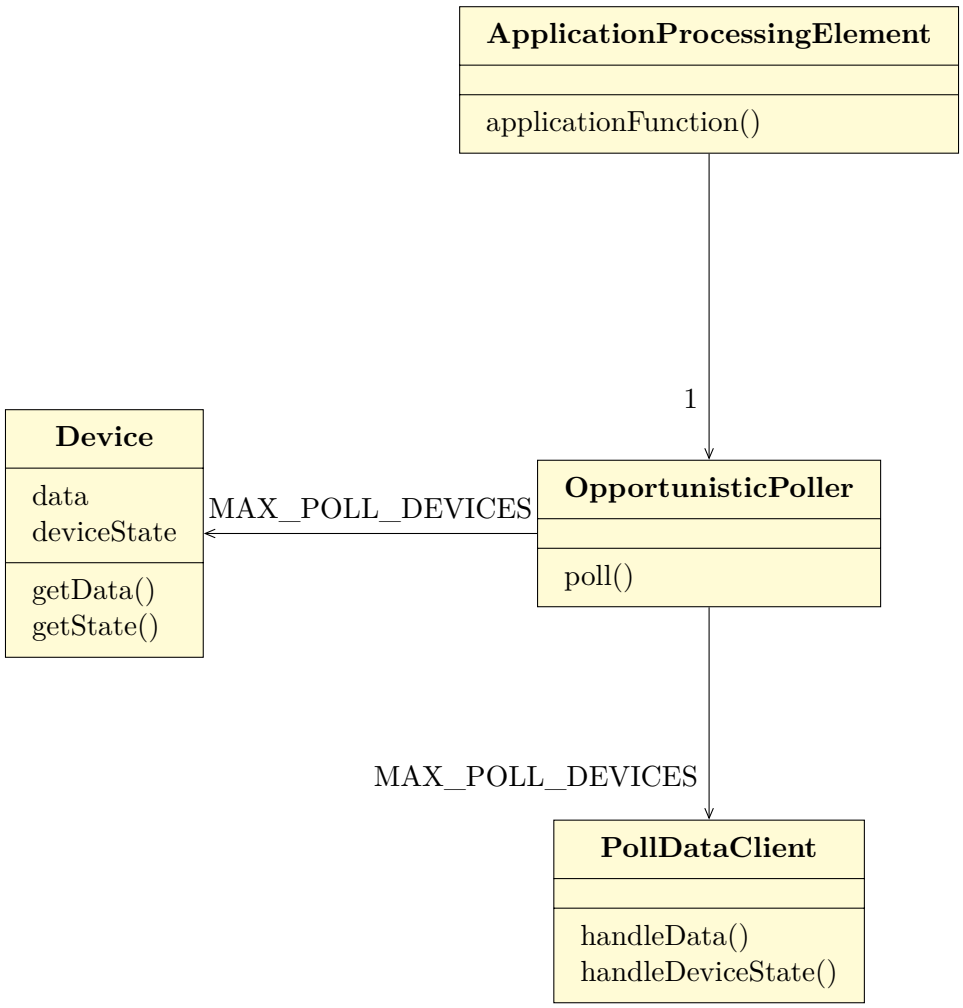
این الگو [۱] یک روش دیگر برای دریافت داده‌ها از سنسورها است و زمانی استفاده می‌شود که استفاده از الگوی Interrupt ممکن نیست یا این که داده‌هایی که می‌خواهیم ضبط کنیم آن قدر اضطراری نیستند و می‌توان برای دریافت آن‌ها صبر کرد. عملکرد این الگو به این صورت است که با سرکشی‌کردن به صورت دوره‌ای داده‌ها را دریافت 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.



شکل ۷: دیاگرام کلاس Periodic Polling

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 Interruptای 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 می‌بینند که لازم است داده‌های جدید از Device گرفته‌شود، این تابع صدا زده می‌شود.

polling ساده‌تر از راه‌اندازی و استفاده از ISR (الگوی Interrupt) است، هرچند Polling دوره‌ای معمولاً با ISR مرتبط با تایمر



شکل ۸: دیاگرام کلاس Opportunistic Polling

Polling انجام می‌شود. Polling می‌تواند وضعیت بسیاری از دستگاه‌ها را همزمان بررسی کند اما معمولاً نسبت به وقفه‌ها کمتر به موقع است. اگر داده‌ها سریع‌تر از زمان Polling برسند، داده‌ها از دست خواهند رفت. این موضوع در بسیاری از برنامه‌ها مشکلی ایجاد نمی‌کند اما در برخی دیگر می‌تواند فاجعه‌بار باشد.

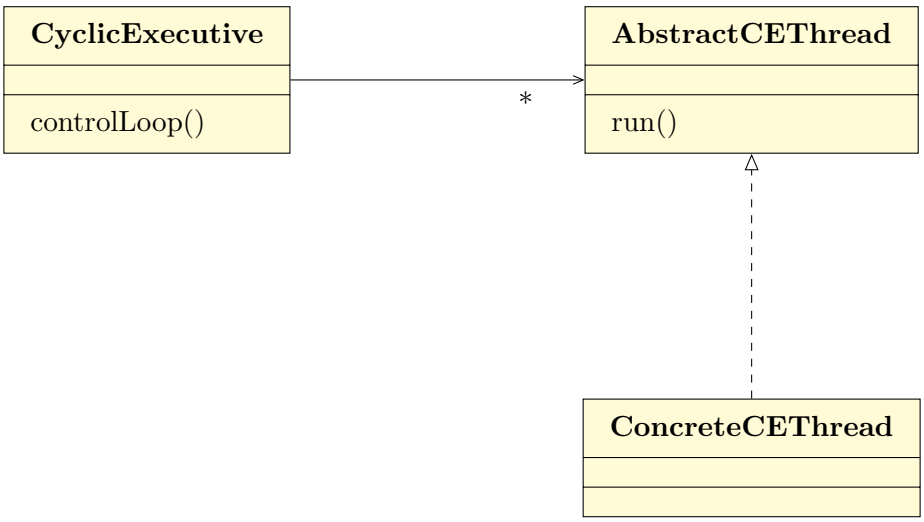
۲.۲ الگوهای طراحی برای همزمانی نهفته و مدیریت حافظه

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 [۱] 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.
کلاس CyclicExecutive با داشتن یک حلقه تکرار همیشگی، تابع run را از هر یک از AbstractCEThreadهایی که وجود دارد صدا می‌زند. [۲]

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



شکل ۹: دیاگرام کلاس Cyclic Executive

این الگو به دلیل سادگی و مصرف کم منابع، برای دستگاه‌های با حافظه کوچک مناسب است. اما برای پاسخ‌دهی به رویدادهای با فوریت بالا ضعیف است و تعامل بین Threadها را پیچیده‌تر می‌کند. بن‌بست‌ها تنها در صورت بروز اشتباه رخ می‌دهند و وظیفه ناسازگار می‌تواند کل سیستم را متوقف کند. در سیستم‌های پیش‌گیرانه، وظایف دیگر حتی با شکست یک وظیفه ادامه می‌یابند.

۲.۲.۲ الگوی **Static Priority**

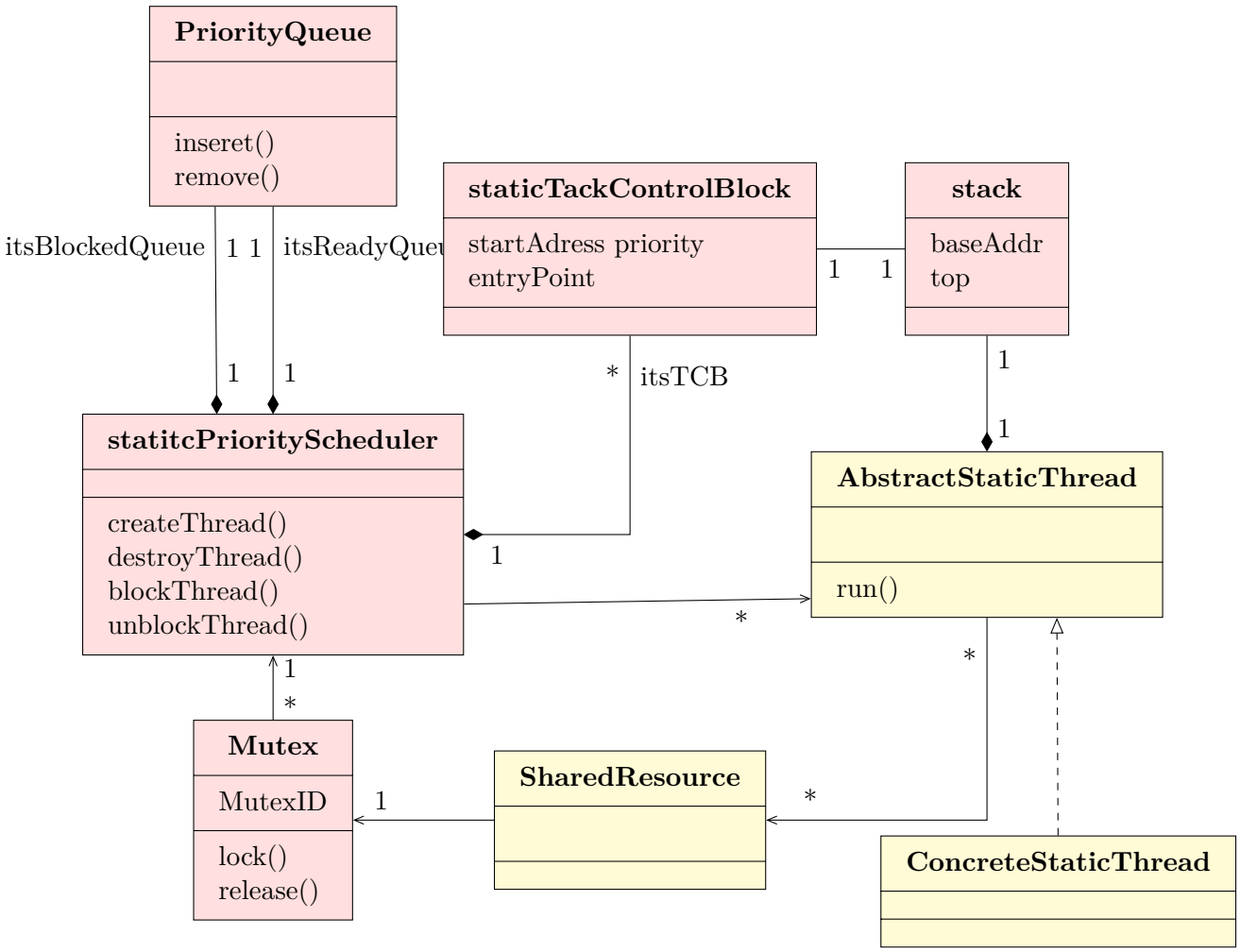
این [۱] یکی از پرکاربردترین الگوهای برنامه‌ریزی در سیستم‌های نهفته بی‌درنگ است. این الگو به ما این قدرت را می‌دهد تا بتوانیم با استفاده از یک سیستم اولویت‌دهی به تسک‌ها، آن‌ها را انجام دهیم. در این سیستم فرض می‌شود که همه تسک‌ها از نوع سنکرون هستند و آن‌ها را بر اساس زمان ددلاینشان اولویت‌دهی می‌کنیم. 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.

با این حال، ممکن است برای سیستم‌های ساده‌تر بدون وقایع ناگهانی اضطراری زیاد باشد. در حالی که تحلیل زمان‌بندی را ساده می‌کند، پیاده‌سازی اشتباه می‌تواند منجر به وارونگی اولویت نامحدود شود وقتی که تسک‌ها، منابع را به اشتراک می‌گذارند. این مشکل را می‌توان با استفاده از الگوهای به اشتراک‌گذاری منابع که وارونگی اولویت را محدود می‌کنند، کاهش داد. روش معمول برای تعیین اولویت‌ها بر اساس مدت زمان ددلاین است، به طوری که ددلاین‌های کوتاه‌تر اولویت بالاتری دریافت می‌کنند. الگوی اولویت ثابت بیشتر بر پاسخگویی تأکید دارد تا عدالت و زمانی که وظایف بیشتر وقت خود را در انتظار شروع شدن توسط وقایع می‌گذرانند، بسیار مؤثر است.

شکل ۱۰ ساختار کلاسی این الگو را نشان می‌دهد. کلاس‌هایی که به رنگ صورتی رسم‌شده‌اند، کلاس‌هایی هستند که توسط یک سیستم‌عامل بی‌درنگ در دسترس قرار می‌گیرند. هر AbstractStaticThread یک StaticTaskControlBlock معادلش دارد که اطلاعات مهمی درباره نحوه برنامه‌ریزی زمانی دارد. کلاس AbstractStaticThread که با کلاس Scheduler ارتباط دارد، رابط run() را برای کلاس ConcreteStaticThread تعریف می‌کند، که دارای اشیایی است تسک‌های سیستم را انجام می‌دهند. کلاس Mutex دسترسی به SharedResourceها را مدیریت می‌کند و Threadها را در صورت لزوم بلوکه می‌کند. PriorityQueue اولویت تسک‌ها را با استفاده از صف بلوکه‌شده‌ها و صف آماده‌ها مدیریت می‌کند. StaticPriorityScheduler با استفاده از StaticTaskControlBlock برای هر Thread، با توجه به اولویت آن، مدیریت انجام آن را انجام می‌دهد.

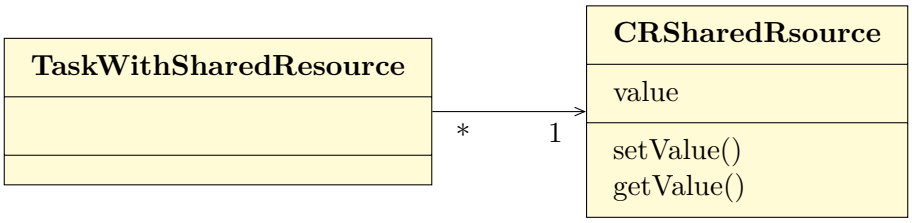
۳.۲.۲ الگوی **Critical Region**

این الگو [۱] 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 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.



شکل ۱۰: دیاگرام کلاس Static Priority

در این الگو باید زمان بحرانی تا حد ممکن کوتاه باشد، زیرا که این الگو می‌تواند زمان‌بندی سایر تسک‌ها را تحت تاثیر قرار دهد. مشکل واونگی اولویت بی‌نهایت در این الگو وجود ندارد زیرا که در زمان بحرانی سوییچ‌کردن بین تسک‌ها را نداریم. یکی از مسائلی که باید در استفاده از این الگو دقت کرد، فراخوانی Critical Region یک تابع از داخل Critical Region یک تابع دیگر است. در چنین صورتی ممکن است که تابع فراخوانی‌شده، فرایند سوییچ‌کردن تسک‌ها را شروع کند در صورتی که تابع فراخوان همچنان در حالت بحرانی است.



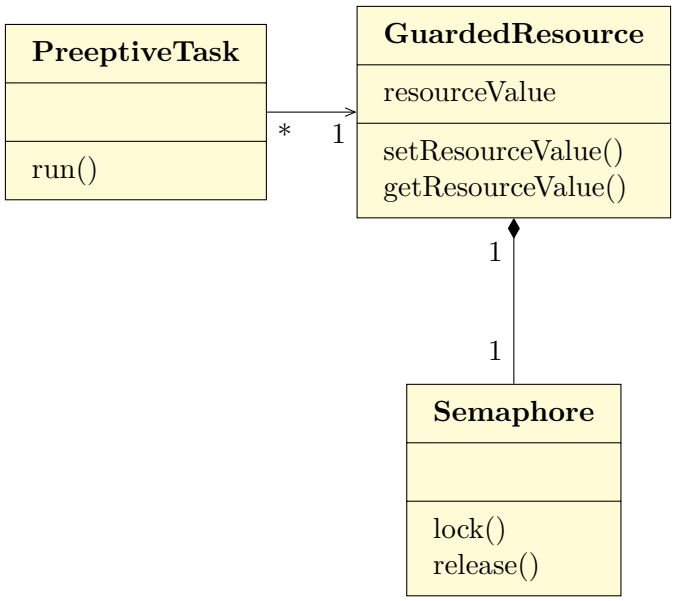
شکل ۱۱: دیاگرام کلاس Critical Region

در شکل ۱۱ کلاس CRSharedRsource باید تنها در دست یک Thread باشد. در این کلاس هر کدام از توابع setValue و getValue باید به صورت جداگانه Critical Region را پیاده‌سازی کنند. کلاس TaskWithSharedResource اطلاعی از مسئله Critical Region ندارد و این مسئله در خود CRSharedRsource هندل‌شده.[۳]

[۳] یک حالت دیگر از این الگو در [۱] بیان شده که در آن، مسئله Shared Resource نیست و صرفا مربوط به زمان‌بندی است. در این حالت خود Task باید مسئله Critical Region را هندل‌کند.

۴.۲.۲ الگوی Guarded Call

این الگو [۱] دسترسی به خدماتی را که ممکن است در صورت استفاده همزمان توسط چندین Thread تداخل کنند، مدیریت می‌کند و از مکانیزم قفل برای جلوگیری از این تداخل استفاده می‌کند. Semaphoreها انحصار متقابل را در محیط چندوظیفه‌ای تضمین می‌کنند و دسترسی به‌موقع به خدمات را هنگامی که توسط وظایف دیگر استفاده نمی‌شوند، فراهم می‌کنند. با این حال، اگر با الگوهای دیگر ترکیب نشود، می‌تواند باعث وارونگی اولویت نامحدود شود. این الگو نیاز به همگام‌سازی یا تبادل داده به‌موقع را برطرف می‌کند و با جلوگیری از دسترسی همزمان، یکپارچگی داده‌ها را تضمین می‌کند. استفاده اشتباه ممکن است منجر به وارونگی اولویت شود، اما راه‌حل‌های پشتیبانی‌شده توسط RTOS می‌توانند این مشکل را کاهش دهند. ساختار کلاسی این الگو در شکل ۱۲ نمایش داده شده‌است.



شکل ۱۲: دیاگرام کلاس Guarded Call

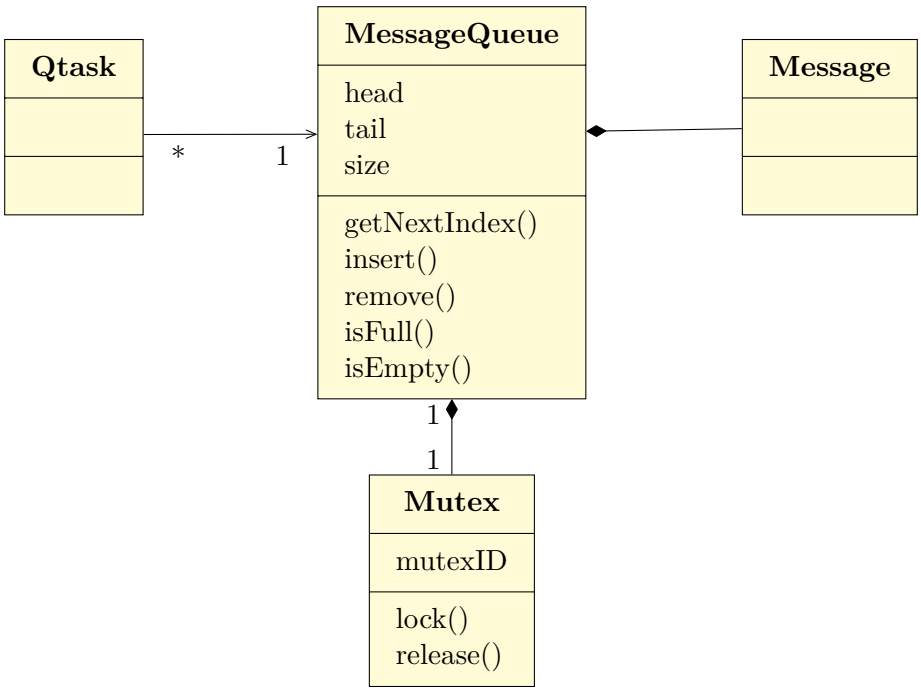
کلاس GuardedResource با در اختیار داشتن Semaphoreها و فراخوانی lock زمانی که یک تسک می‌خواهد از آن استفاده کند، متوجه می‌شود که آیا باید به تسک اجازه استفاده از منابع را بدهد یا خیر. زمانی تسک فعالی که در حال استفاده از GuardedResource است، کارش تمام می‌شود، GuardedResource تابع release را برای Semaphore مربوط به آن دسته تسک‌ها را صدا می‌زند و اکنون منبع آزاد است تا زمانی که یک تسک جدید آن را قفل کند. دقت شود که در ساختار این الگو کلاس‌هایی که توسط RTOS ارائه می‌شوند رسم نشده و صرفا هدف نشان‌دادن رفتار الگو است.

۵.۲.۲ الگوی Queuing

این الگو [۱] 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.

ساختار این الگو به این شکل است که هر QTask برای ارسال یا دریافت پیام، از MessageQueue استفاده می‌کند. این کلاس با فراهم‌سازی توابعی مانند insert و remove، این امکان را فراهم می‌سازد.

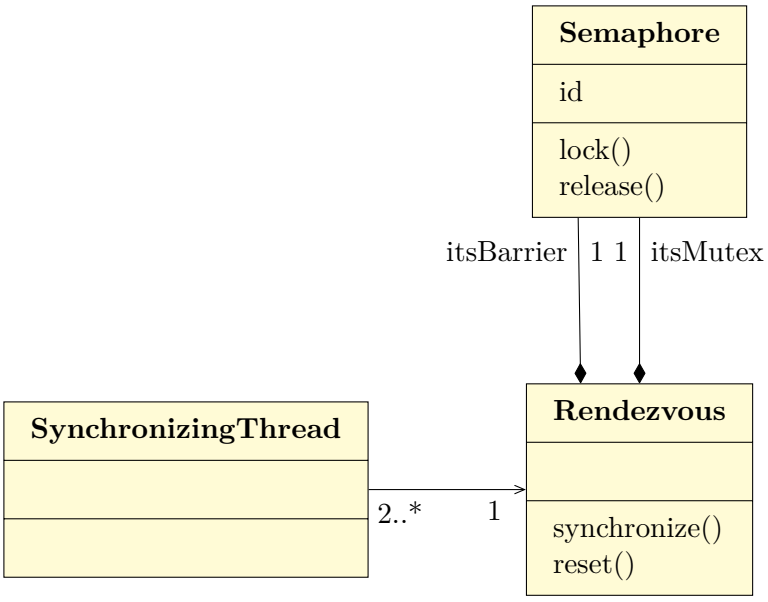
این الگو یک روشی برای استفاد سریال از داده‌ها فراهم می‌کند. وجود Mutex در این الگو باعث می‌شود که خود MessageQueue آلوده نشود. به دلیل استفاده از ارتباطات آسنکرون، دریافت داده‌ها در این الگو سریع‌تر از الگوی Guarded Call است. یکی از نکاتی که باید به آن توجه کرد، انتخاب درست اندازه صف است. اگر صف خیلی کوچک باشد، داده‌ها از دست می‌روند و اگر خیلی بزرگ باشد، استفاده از حافظه غیر بهینه است.



شکل ۱۳: دیاگرام کلاس Queuing

۶.۲.۲ الگوی Rendezvous

این الگو [۱] 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.



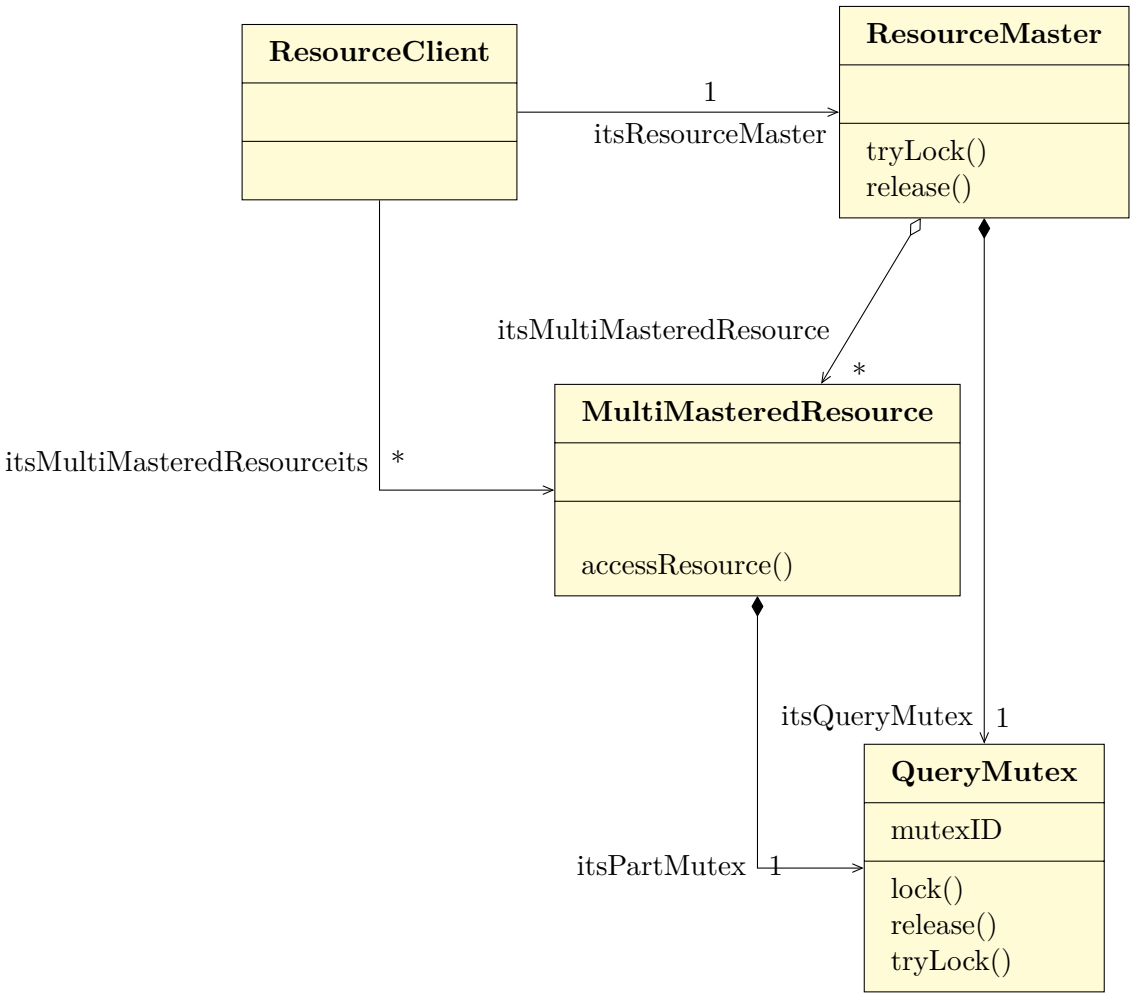
شکل ۱۴: دیاگرام کلاس Rendezvous

همانطور که در شکل ۱۴ دیده می‌شود، کلاس SynchronizingThread با فراخوانی تابع synchronize از Rendezvous

می‌خواهد خود را سنکرون کند. این عملیات در Rendezvous با بررسی چندین شرط انجام می‌شود و در صورتی که شروط برقرار نباشند، این Thread باید بلوکه شود. این الگو در عین سادگی، منعطف و قابل تطبیق است. این الگو زمانی کاربرد دارد که می‌خواهیم تسک‌ها در زمان سنکرون‌شدن، متوقف شوند. اگر بخواهیم در توقف اتفاق نیافتد، باید این الگو را با الگوهای دیگر ترکیب کنیم.

۷.۲.۲ الگوی Simultaneous Locking

این الگو [۱] بر جلوگیری از بن‌بست تمرکز دارد و اطمینان حاصل می‌کند که همه منابع مورد نیاز به‌طور همزمان قفل می‌شوند یا هیچ‌کدام قفل نمی‌شوند. این روش با شکستن شرط نگه‌داشتن برخی منابع در حین انتظار برای دیگران، از وقوع بن‌بست جلوگیری می‌کند، اما ممکن است باعث افزایش تأخیر در اجرای وظایف شود. این الگو به وظایف با اولویت بالاتر اجازه می‌دهد در صورت نیاز نداشتن به منابع قفل‌شده اجرا شوند. با این حال، این الگو وارونگی اولویت را حل نمی‌کند و ممکن است آن را بدتر کند مگر با استفاده از الگوهای دیگر در کنار این الگو مانند الگوی Priority Inheritance.

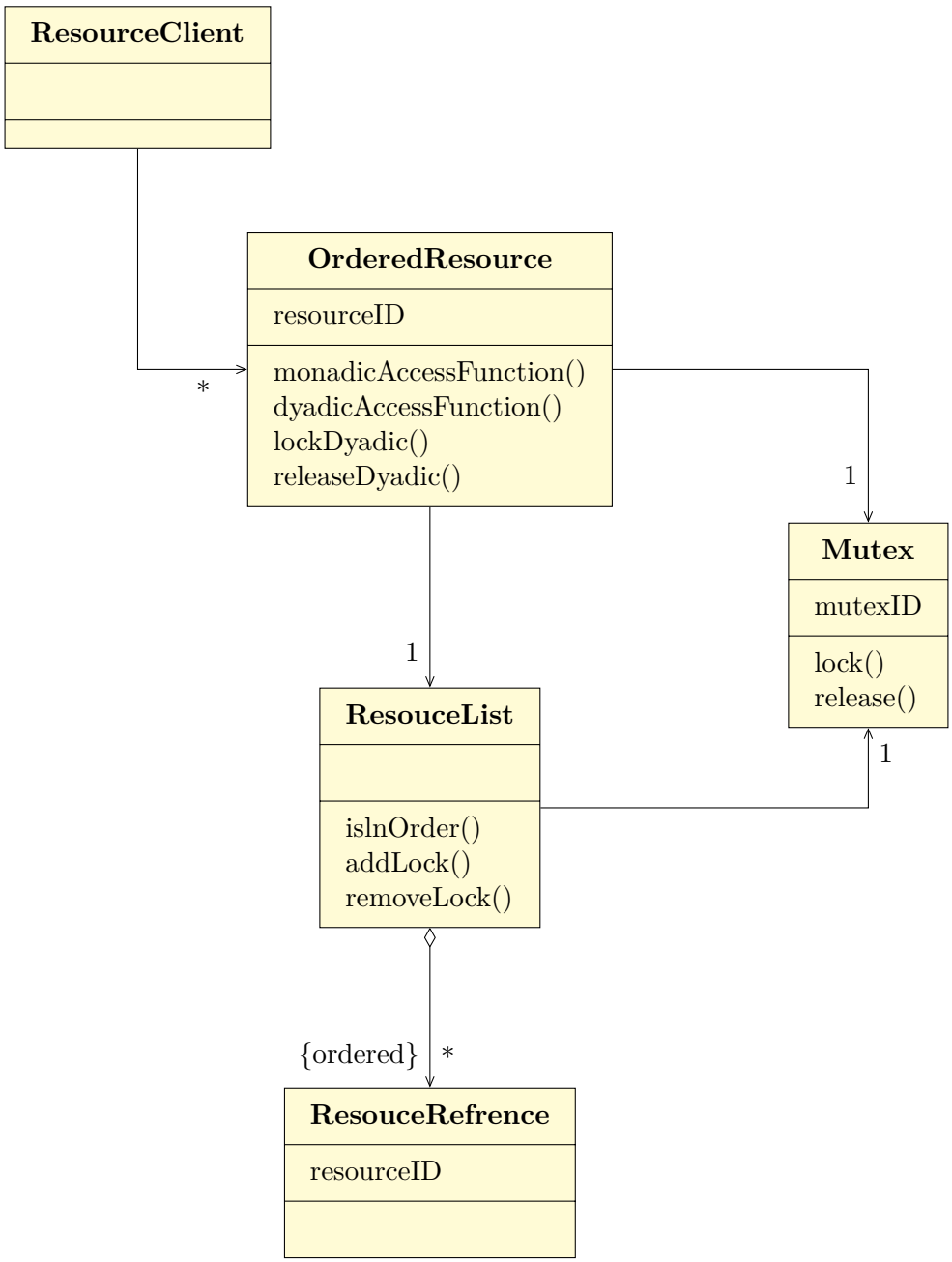


شکل ۱۵: دیاگرام کلاس Simultaneous Locking

همانطور که در شکل ۱۵ دیده می‌شود، هر ResourceClient سرویس‌گیرنده از MultiMasteredResource است. در این ساختار، این کلاینت‌ها حق استفاده از منابع را تا زمانی که نتوانسته‌اند کنترل ResourceMaster را بدست بگیرند، ندارند. زمانی که درخواست قفل‌کردن برای ResourceMaster می‌آید، این کلاس سعی می‌کند که تمامی منابع را قفل کند، اگر در انجام این کار موفق بود، کلاینت اجازه دارد از منابع استفاده کند. در غیر این صورت، تمامی منابعی که توسط ResourceMaster قفل شده‌بودند، دوباره آزاد می‌شوند و یک کد خطا به کلاینت برگردانده می‌شود.

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.



شکل ۱۶: دیاگرام کلاس Ordered Locking

همانطور که در شکل ۱۶ دیده می‌شود، در این ساختار، کلاینت باید برای استفاده از منابع، از کلاس OrderedResource استفاده کند. این کلاس می‌تواند منابع را به دو فرم monadic و dyadic در اختیار او قرار دهد. این کلاس با مدیریت منابع، این اطمینان را ایجاد می‌کند که ترتیب قفل‌شدن و آزادسازی منابع، طبق ترتیبی که از پیش بر اساس resourceID تنظیم‌شده انجام می‌شود.

۳.۲ الگوهای طراحی برای ماشین‌های حالت

این دسته از الگوها که توسط Douglass در [۱] معرفی شده‌است، الگوهایی هستند که براساس ماشین‌های حالت ساخته شده‌اند.

۱.۳.۲ الگوی Single Event Receptor

این الگو [۱] 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. در این برای تمامی ماشین حالت، تنها یک دریافت‌کنند رویداد داریم که این باعث محدودیت در Scalability سیستم می‌شود. اما حسن بزرگ این الگو این است که برای هر دو حالت رویدادهای سنکرون و آسنکرون قابل استفاده است.

۲.۳.۲ الگوی Multiple Event Receptor

در این الگو [۱]، برای هر یک از رویدادهای ممکن که توسط کلاینت رخ می‌دهد، یک دریافت‌کنند مجزا داریم. این الگو تنها برای رویدادهای سنکرون کاربرد دارد. این الگو با شکستن منطق حالت‌های برنامه به تعدادی دریافت‌کننده رویداد، پیاده‌سازی الگو را ساده می‌کند. اما عیب آن این است که تنها رویدادهای سنکرون را پشتیبانی می‌کند.

۳.۳.۲ الگوی State Table

این الگو [۱] 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. در این الگو، ساختارهای AND-State نیز به صورت ساختاری flat پیاده‌سازی می‌شود. این به این معنا است که اگر به طور مثال ۲ ماشین حالت مستقل، هرکدام با ۳ حالت داشته‌باشیم، این الگو این ساختار را به یک ماشین حالت به فرم ۳×۳ حالت تبدیل می‌کند.

۴.۳.۲ الگوی State

این الگو [۱] 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. در این ساختار، با اضافه‌شدن هر حالت جدید، باید یک کلاس جدید تعریف شود. این الگو نسبت به الگوی State Table حافظه بیشتری اشغال می‌کند اما با توزیع وظیفه مدیریت حالت بین کلاس‌های مختلف، پیاده‌سازی ساده‌تری دارد. یکی دیگر از مزیت‌های استفاده از این الگو، این است که این کلاس‌های حالت را می‌توان بین کلاینت‌های مختلف به اشتراک گذاشت.

۵.۳.۲ الگوی Decomposed And State

این الگو [۱]، با استفاده از الگوی State می‌خواهد AND-Stateها را با ایجاد همکاری میان اشیاء مختلف که هر کدام یک AND-State را مدیریت می‌کند، مدیریت کند. شیء اصلی، ماشین حالت کلی را نظارت می‌کند، در حالی که اشیاء دیگر AND-Stateهای تکی را مدیریت می‌کند. این الگو به پیاده‌سازی ماشین‌هایحالت با منطق‌های مستقل می‌پردازد و طراحی AND-Stateها را حفظ می‌کند. این الگو مدیریت حالت را از طریق واگذاری مسئولیت‌ها ساده می‌کند، اما نیاز به مدیریت دقیق لیست‌ها و اشاره‌گرها دارد تا از بروز خطاهای نامشخص جلوگیری شود.

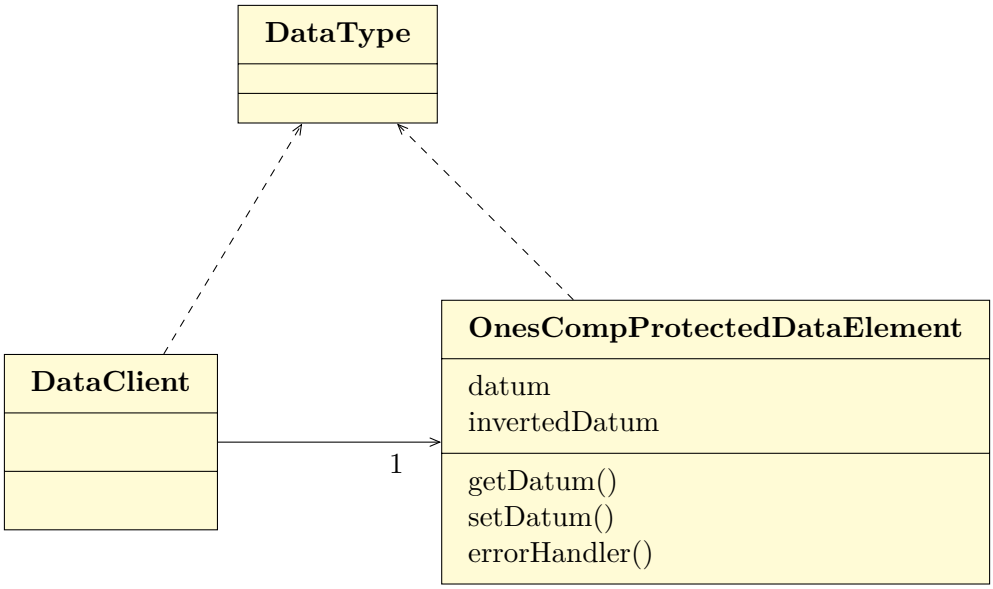
۴.۲ الگوهای امنیت و قابلیت اطمینان

سیستم‌هایی که به طور حیاتی باید امنیت و قابلیت اطمینان بالایی داشته باشند، دسته مهمی از سیستم‌ها هستند که در [۱] یک فصل مجزا برای الگوهای مرتبط با آن‌ها در نظر گرفته‌شده. این الگوها به طور کلی با اضافه‌کردن تکرار به سخت‌افزار و نرم‌افزار، به قابلیت اطمینان و امنیت سیستم کمک می‌کنند. در ادامه این الگوها را می‌بینیم.

۱.۴.۲ الگوی One’s Complement

این الگو [۱] 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.

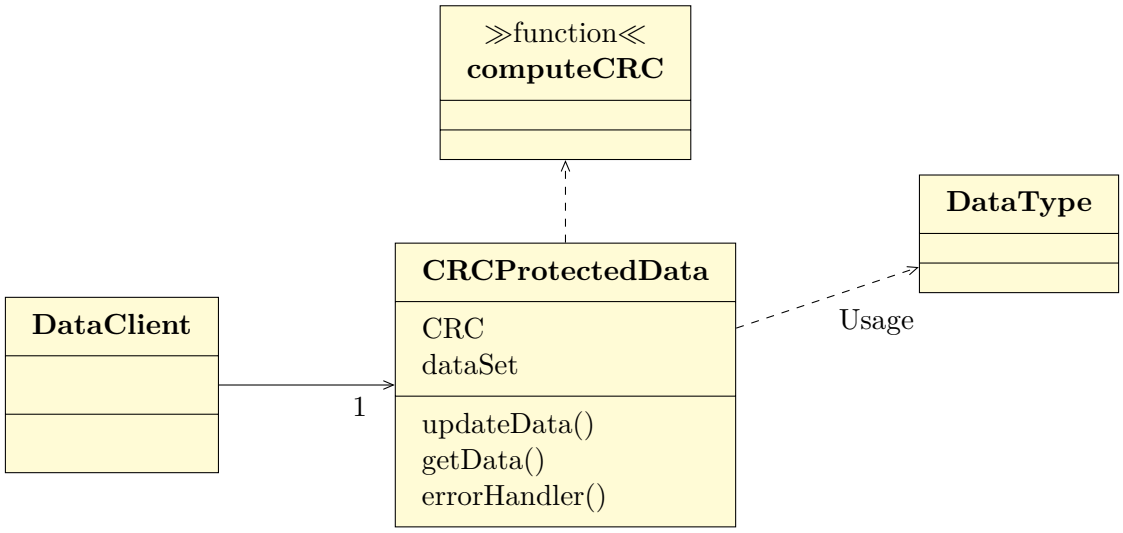


شکل ۱۷: دیاگرام کلاس One's Complement

همانطور که در شکل ۱۷ دیده می‌شود، کلاس OnesCompProtectedDataElement با انجام عملیات تکرار در ذخیره‌سازی و بررسی داده و مقایسه آن با مقدار قرینه آن در زمان خواندن، فرایند الگو را داخل خود انجام می‌دهد. این الگو به دلیل تکرار در نوشتن داده‌ها، باعث استفاده دو برابر از حافظه می‌شود. همچنین این دو داده نوشته‌شده در زمان خوانده‌شدن نیز باعث سربار اضافی می‌شوند. با این حال این الگو می‌تواند خطاهای گفته‌شده را تشخیص دهد.

۲.۴.۲ الگوی CRC

این الگو [۱] با استفاده از یک کد باینری با طول ثابت CRC، یک الگوریتم خطایابی ارائه می‌دهد که برای ساختار داده‌های بزرگ بسیار کاربرد خواهد داشت.



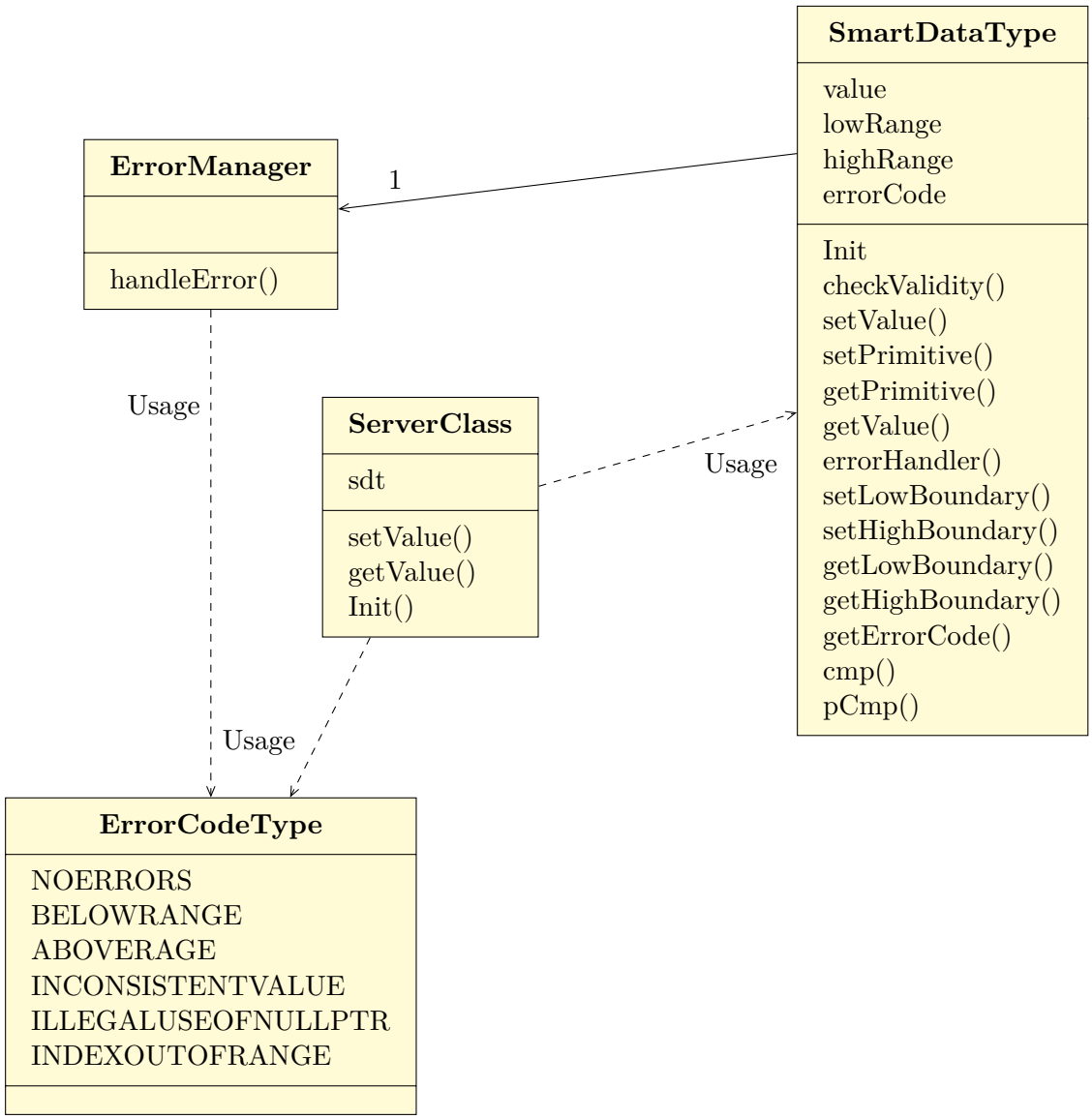
شکل ۱۸: دیاگرام کلاس CRC

همانطور که در شکل ۱۸ مشخص است، این الگو با استفاده از یک CRCProtectedData، می‌تواند پیاده‌سازی مورد نظر را انجام دهد. این کلاس با استفاده از تابع مربوطه برای محاسبه CRC، زمانی که داده‌ها می‌خواهند ذخیره شوند، این مقدار را نیز ذخیره می‌کند.

همچنین زمان خواندن داده، با محاسبه دوباره CRC و مقایسه آن با مقدار قبلی، از صحت عملکرد مطمئن می‌شود و در صورت وجود مشکل، فرایند هندل‌کردن خطا باید انجام شود. این الگو با معرفی حجم کمی از داده اضافی در کنار داده اصلی، برای تشخیص خطاهای تک‌بیت روش خوبی ارائه می‌دهد. این الگو بیشتر در حوزه ارتباطات استفاده می‌شود و زمانی که می‌خواهیم یک پیام را بفرستیم، برای تشخیص خطاهای احتمالی، از این الگو استفاده می‌شود.

۳.۴.۲ الگوی Smart Data

این الگو [۱] 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.

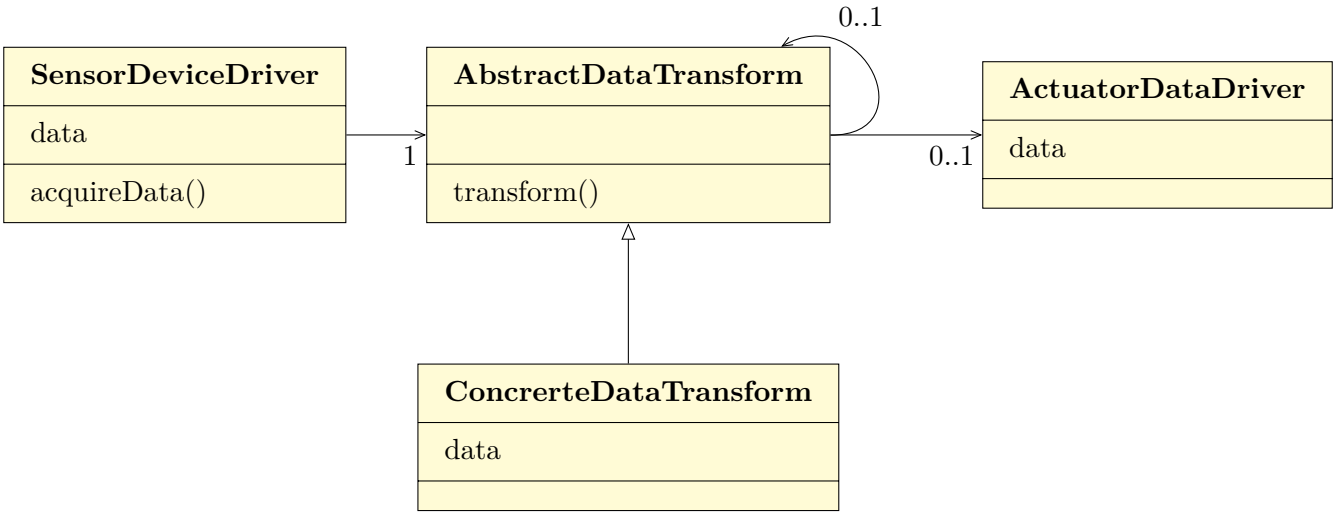


شکل ۱۹: دیاگرام کلاس Smart Data

همانطور که در شکل ۱۹ دیده می‌شود، ساختار داده SmartDataType، توابع زیادی برای چک‌کردن داده خود به ServerClass ارائه داده‌است. این الگو با قراردادن چک‌های امنیتی بر روی ساختار داده، می‌تواند باعث امنیت داده در زمان setشدن شود. اما این چک‌ها خود سربار برای انجام هر عملیات با این داده‌ها می‌شوند.

۴.۴.۲ الگوی Channel

الگوی کانال [۱] از تکرار در مقیاس متوسط تا بزرگ پشتیبانی می‌کند و به شناسایی و مدیریت خطاهای زمان اجرا کمک می‌کند. یک کانال داده‌ها را از مرحله دریافت تا Actuation پردازش می‌کند و واحدی مستقل و خودکفا از عملکرد را فراهم می‌کند. ارزش این الگو در استفاده از چندین کانال برای رفع نگرانی‌های ایمنی و قابلیت اطمینان است. این الگو شناسایی واضح خطا و تداوم خدمات یا دستیابی به حالت Fault-Safe را ارائه می‌دهد. با این حال، به حافظه، زمان پردازش و سخت‌افزار اضافی نیاز دارد.



شکل ۲۰: دیاگرام کلاس Channel

همانطور که در شکل ۲۰ دیده می‌شود، این الگو با ایجاد زنجیره‌ای از پردازش‌ها (DataTransform)، داده‌ها را از سنسور دریافت کرده و در هر مرحله پردازشی روی آن انجام می‌دهد. در نهایت داده‌های پردازش‌شده به سمت Actuatorها هدایت می‌شود.

۵.۴.۲ الگوی Protected Single Channel

این الگو [۱] 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 تکرار 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.

ساختار این الگو در شکل ۲۱ آمده است. این الگو نسبت به الگوی Channel تکرار را کاهش می‌دهد ولی در عوض تعدادی چک در مراحل مختلف پردازش داده‌ها اضافه می‌کند که به امنیت سیستم کمک می‌کند.

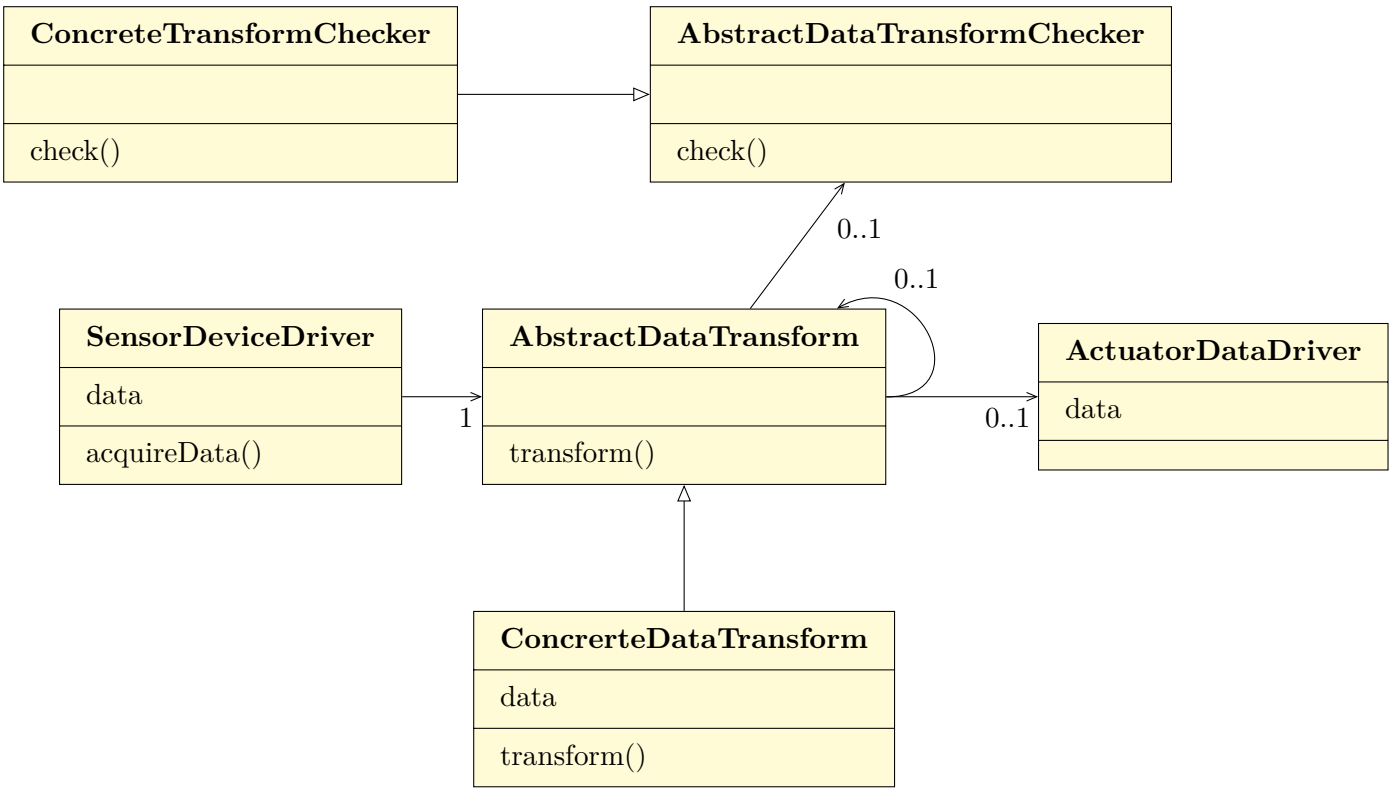
۶.۴.۲ الگوی Dual Channel

این الگو [۱] با ایجاد چند کانال و ایجاد تکرار در سطحی بالاتر، امنیت را تحقق می‌بخشد. اگر کانال‌های مورد استفاده از یک نوع باشند، این الگو الگوی Homogeneous Redundancy و در غیر این صورت الگوی Heterogeneous Redundancy 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.

همانطور که در شکل ۲۲ مشخص است، این الگو کاملا مشابه ساختار الگوی Protected Single Channel است با این تفاوت که کلاس ConcreteTransformChecker به دو کانال متصل است. یکی کانل فعلی یا خود. و دیگری کانال یدکی یا دیگر. در صورتی که در یک کانال خطا رخ دهد، این الگو می‌تواند این کانال را غیرفعال کرده و کانال دیگر را فعال کند.

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.



شکل ۲۱: دیاگرام کلاس Protected Single Channel

۱.۵.۲ الگوی Layered

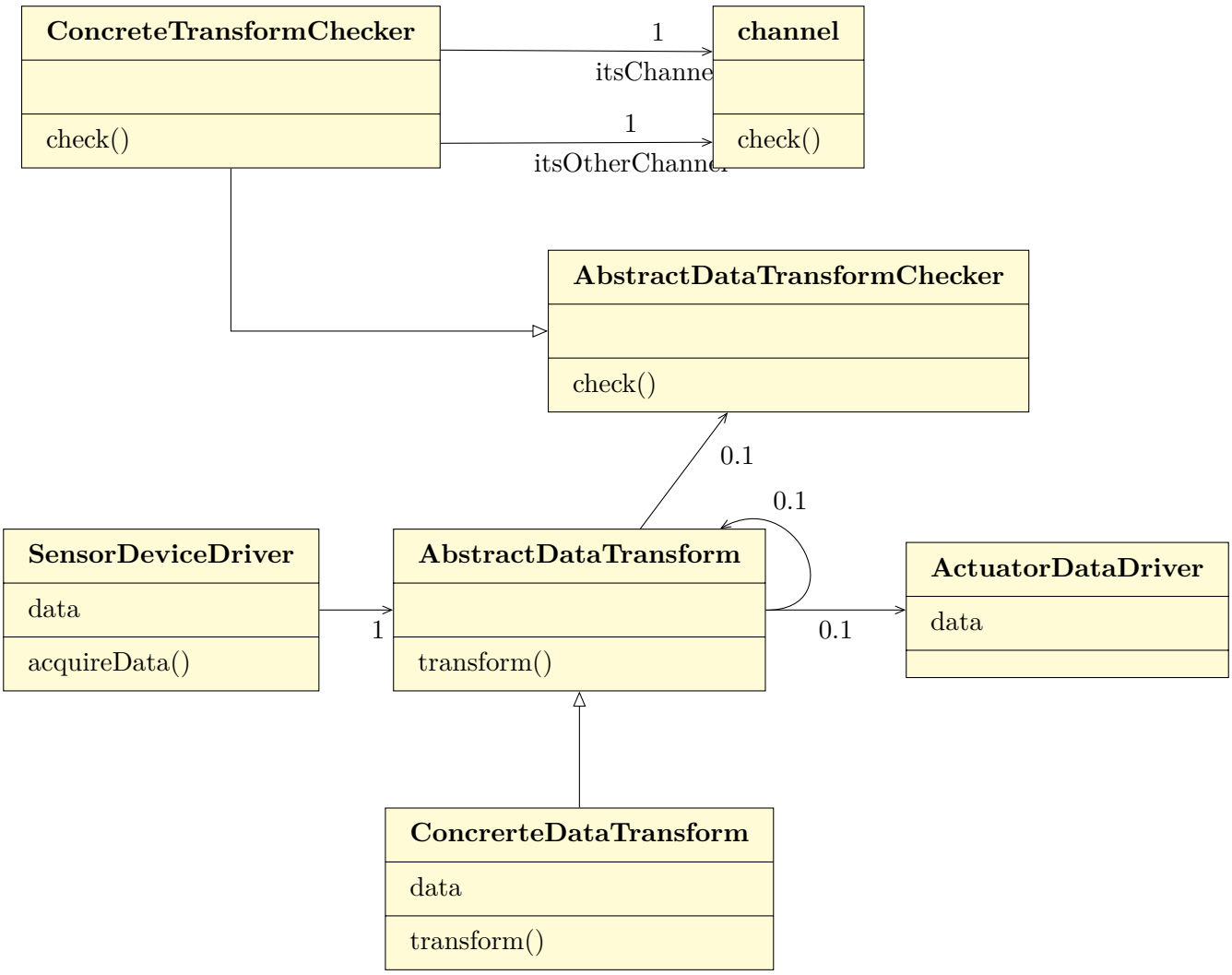
الگوی لایه‌ای [۴] دامنه‌های سیستم را بر اساس سطوح انتزاعی مختلف به صورت سلسله‌مراتبی سازمان‌دهی می‌کند. مفاهیم انتزاعی‌تر در یک دامنه با استفاده از مفاهیم ملموس‌تر در دامنه‌های دیگر پیاده‌سازی می‌شوند. این ساختار به متخصصان اجازه می‌دهد تا به طور موثر در زمینه تخصصی خود کار کنند بدون اینکه نیاز به درک 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 [۴] 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 [۴] 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.



شکل ۲۲: دیاگرام کلاس Dual Channel

۴.۵.۲ الگوی **Channel**

الگوی معماری کانال [۴] 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 تکرار 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**

این الگو [۴] 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 محول 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 سطوح مختلفی از جزئیات رفتار سیستم را ارائه می‌دهد.

۶.۵.۲ الگوی Hierarchical Control

الگوی کنترل سلسله‌مراتبی [۴] 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 [۴] 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 در 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 Component [۴] معماری سیستم را قوی و قابل استفاده مجدد می‌سازد 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 [۴] [۴] 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 متدولوژی 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 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.

[۴] Real-time Object Oriented Methodology

۱.۶.۲ الگوی Message Queuing

الگوی صف پیام [۴] 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 پاس داده می‌شوند و از مسائل فساد داده‌ای که در سیستم‌های همزمان رایج است، جلوگیری می‌شود. 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.)

۲.۶.۲ الگوی Interrupt

وقفه‌ها [۴] 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 [۴] 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 [۴] 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 [۴] در سیستم‌های کوچک یا سیستم‌هایی که نیاز به اجرای قابل پیش‌بینی دارند، به طور گسترده‌ای استفاده می‌شود. این الگو با سادگی و پیش‌بینی‌پذیری خود، پیاده‌سازی آسانی دارد و برای محیط‌های محدود به حافظه که استفاده از یک سیستم‌عامل بی‌درنگ به طور 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 [۴] 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 ددلاین‌های 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 [۴]، 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 [۴] 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 جلوگیری از تکه‌تکه شدن حافظه و بهبود پیش‌بینی‌پذیری است.

۲.۷.۲ الگوی Pool Allocation

الگوی Pool Allocation [۴] 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 [۴] 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 [۴] 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 [۴] 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 [۴] 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 [۴] 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 ددلاین‌ها 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 [۴] 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 کلاینت‌های 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 [۴] 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

توزیع یک جنبه مهم در معماری است که سیاست‌ها، رویه‌ها و ساختار سیستم‌هایی را که ممکن است به طور همزمان در چند فضای آدرس وجود داشته باشند، تعریف می‌کند. معماری‌های توزیع به دو نوع اصلی نامتقارن و متقارن تقسیم می‌شوند. در معماری نامتقارن، اتصال اشیا به فضاهای آدرس در زمان طراحی مشخص است، در حالی که در معماری متقارن این اتصال تا زمان اجرا مشخص نمی‌شود. این بخش از [۴] به بررسی معماری‌های توزیع در سطح برنامه کاربردی می‌پردازد و تمرکز آن بر چگونگی یافتن و ارتباط اشیا با یکدیگر است.

۱.۹.۲ الگوی Shared Memory

این الگو [۴] به پردازنده‌های متعدد امکان می‌دهد داده‌ها را با استفاده از یک ناحیه حافظه مشترک که معمولاً توسط چیپ‌های RAM چند پورت تسهیل می‌شود، به اشتراک بگذارند. این الگو ساده و مفید است وقتی که نیاز به اشتراک‌گذاری داده‌ها بین پردازنده‌ها وجود دارد، بدون نیاز به پاسخ‌های فوری به پیام‌ها یا رویدادها. این راه‌حل معمولاً ترکیبی از سخت‌افزار و نرم‌افزار است، به طوری که سخت‌افزار با ارائه Semaphoreهای تک‌سیکل CPU و دسترسی به حافظه از درگیری‌ها جلوگیری می‌کند، در حالی که نرم‌افزار دسترسی مطمئن را تضمین می‌کند. برای داده‌های فقط خواندنی، ممکن است چنین مکانیزم‌های همزمانی لازم نباشد.

این الگو برای سیستم‌هایی که نیاز به اشتراک‌گذاری مقادیر زیادی از داده‌ها به صورت پایدار بین پردازنده‌ها دارند، مانند پایگاه‌های داده مشترک یا کد اجرایی، ایده‌آل است. این الگو معمولاً در کاربردهایی که سخت‌افزار و نرم‌افزار به طور مشترک طراحی می‌شوند، استفاده

می‌شود و نه راه‌حل‌های تجاری آماده. سخت‌افزار ممکن است تعداد بلوک‌های قفل‌پذیر را محدود کند و نرم‌افزار باید Semaphoreها را برای کنترل دسترسی مدیریت کند و شرایط رقابت را برای اطمینان از نوشتن موفق بررسی کند.
حافظه مشترک برای ذخیره داده‌های بزرگ که به چندین پردازنده با سربار کم قابل دسترسی هستند، موثر است، اگرچه ممکن است تحویل پیام‌ها به موقع نباشد زیرا کلاینت‌ها حافظه مشترک را برای به‌روزرسانی‌ها بررسی می‌کنند.

۲.۹.۲ الگوی Remote Method Call

روش‌های Remote Method Call (RMCs) [۴] در سیستم‌های بی‌درنگ نهفته مشابه Remote Procedure Call (RPCs) هستند و امکان فراخوانی سرویس‌های همزمان بین پردازنده‌ها را فراهم می‌کنند. این روش‌ها مانند فراخوانی متدهای محلی عمل می‌کنند؛ به این صورت که کلاینت یک سرویس را بر روی سرور فراخوانی می‌کند و تا زمان تکمیل عملیات در حالت مسدود شده منتظر می‌ماند. RMCها که توسط سیستم‌عامل‌های مختلف پشتیبانی می‌شوند، ارتباطات بین فرآیندی (IPC) را به صورت انتزاعی‌تر فراهم کرده و ارتباطات کلاینت-سرور را بر روی شبکه‌ها ساده‌تر می‌کنند. اگرچه تأخیرهای شبکه و پیچیدگی‌های مدیریت خطا ذاتی هستند، RMCها نسبت به IPC سنتی فرایند را ساده‌تر می‌کنند. انتخاب پروتکل انتقال مناسب، مانند TCP به جای UDP غیرقابل‌اعتماد، می‌تواند نگرانی‌های مربوط به به‌موقع‌بودن و قابلیت اطمینان را برطرف کند.

۳.۹.۲ الگوی Observer

این الگو همان الگوی Observer است که در [۱] گفته شده‌است؛ اما در این‌جا تعریف خود را به کاربردها ارتباط با سنسور محدود نکرده‌است و در ساختار سیستم‌های توزیع‌شده معنا پیدا می‌کند. در اینجا ساختار الگو کاملا مشابه الگوی Observer است؛ اما کلاس‌های کلاینت، هر کدام می‌توانند به صورت توزیع‌شده در آدرسی متفاوت قرار داشته باشند.

۴.۹.۲ الگوی Data Bus

این الگو [۴] الگوی Observer را با ارائه یک گذرگاه مشترک گسترش می‌دهد که در آن چندین سرور اطلاعات خود را منتشر می‌کنند و چندین کلاینت رویدادها و داده‌ها را دریافت می‌کنند. این الگو برای سیستم‌هایی که سرورها و کاربران زیادی باید داده‌ها را به اشتراک بگذارند مناسب است و توسط گذرگاه‌های سخت‌افزاری مانند گذرگاه CAN پشتیبانی می‌شود. گذرگاه داده (Data Buss) به عنوان یک مرکز مشترک برای به اشتراک‌گذاری داده‌ها در پردازنده‌ها عمل می‌کند و به کلاینت‌ها اجازه می‌دهد که داده‌ها را دریافت‌کنند یا برای دریافت آنها مشترک شوند. این الگو به عنوان یک پروکسی با یک مخزن داده متمرکز عمل می‌کند و می‌تواند اشیاء داده مختلف را مدیریت کند. گذرگاه داده بسیار قابل گسترش است و انواع داده‌های جدید را می‌توان بدون تغییر ساختار اصلی در زمان اجرا اضافه کرد. با این حال، مکان گذرگاه داده باید از پیش تعیین شده باشد و مدیریت ترافیک آن ممکن است ظرفیت گره را برای انجام کارهای دیگر محدود کند. این الگو برای معماری‌های متقارن که سرورها در پردازنده‌های کمتر قابل دسترس قرار دارند، مؤثر است.

۵.۹.۲ الگوی Proxy

الگوی پروکسی [۴] با استفاده از یک کلاس جایگزین، سرور واقعی را از کلاینت انتزاع می‌کند و جداسازی و پنهان‌سازی ویژگی‌های خاص سرور از کلاینت‌ها را امکان‌پذیر می‌سازد. این الگو در سیستم‌های نهفته که سرورها ممکن است در فضای آدرس‌های مختلف باشند بسیار مفید است و به کلاینت‌ها اجازه می‌دهد بدون اطلاع از مکان سرور با آن تعامل کنند. این انتزاع طراحی مشتریان را ساده می‌کند و تغییرات سیستم را بدون تغییر در تعاملات کلاینت-سرور تسهیل می‌کند. الگوی پروکسی به مدیریت شفافیت ارتباطات کمک کرده و روش تماس با سرورهای راه دور را محصور می‌کند. این الگو ترافیک ارتباطات را با کاهش تعداد پیام‌های ارسال شده در شبکه و استفاده از سیاست اشتراک برای انتقال داده کاهش می‌دهد.

۶.۹.۲ الگوی Broker

الگوی بروکر [۴] یک نسخه متقارن از الگوی Proxy است که برای شرایطی طراحی شده که مکان کلاینت و سرورها در زمان طراحی مشخص نیست. این الگو یک بروکر را معرفی می‌کند، که یک مخزن ارجاع شیء است و برای هر دو کلاینت و سرورها قابل مشاهده است و به کلاینت‌ها در یافتن سرورها کمک می‌کند. این کار، امکان استقرار معماری‌های متقارن مانند تعادل بار پویا را فراهم می‌کند. الگوی بروکر مسائل شفافیت ارتباطات را حل کرده و نیاز به دانش قبلی از مکان سرورها را از بین می‌برد، که به افزایش مقیاس‌پذیری سیستم و پنهان‌سازی جزئیات زیرین پردازنده‌ها و ارتباطات کمک می‌کند. اگرچه Object Request Brokerهای تجاری به خوبی از این الگو پشتیبانی می‌کنند، اما ممکن است منابع بیشتری نسبت به سیستم‌های کوچکتر نیاز داشته باشند، که در این موارد می‌توان از ORBهای کوچکتر یا پیاده‌سازی سفارشی استفاده کرد.

۱۰.۲ الگوهای معماری امنیت و قابلیت اطمینان

همانطور که در بخش‌های قبلی گفته‌شده، یکی از مهم‌ترین مسائل در سیستم‌های نهفته، مسئله امنیت (Safety) و قابلیت اطمینان (Reliability) است. در این بخش به الگوهایی که در [۴] گفته‌شده می‌پردازیم.

۱.۱۰.۲ الگوی **Protected Single Channel**

redundancy کامل در سیستم‌هایی که امنیت در آن‌ها حیاتی است، پرهزینه است، هم در تکرار سخت‌افزار و هم در توسعه آن. این الگو [۴] یک جایگزین سبک برای افزایش ایمنی و قابلیت اطمینان است که با افزودن چک‌های اضافی و مقداری سخت‌افزار اضافی این کار را انجام می‌دهد. این الگو از یک کانال برای حسگر و تحریک استفاده می‌کند و خطاهای گذرا را شناسایی و مدیریت می‌کند، اما خطاهای پایدار را نمی‌تواند مدیریت کند. این رویکرد از نظر هزینه‌های تکراری و توسعه مقرون به صرفه است و برای سیستم‌هایی که نیاز به عملکرد در حضور خطاهای پایدار ندارند یا حساس به هزینه هستند، مناسب است. با این حال، به دلیل نقاطی که یک خطای منفرد می‌تواند باعث از دست رفتن کل سیستم شود، برای همه سیستم‌های مرتبط با ایمنی مناسب نیست. این الگو مشابه الگوی Protected Single Channel است که در [۱] آورده شده‌است؛ با این تفاوت که در اینجا در ابعاد معماری تعریف شده.

۲.۱۰.۲ الگوی **Homogeneous Redundancy**

این الگو [۴] با استفاده از چندین کانال برای انجام وظایف، قابلیت اطمینان سیستم را بهبود می‌بخشد. این کانال‌ها می‌توانند به صورت متوالی یا به صورت موازی عمل کنند. این الگو از سیستم در برابر خطاهای تصادفی محافظت می‌کند و در صورت خرابی به کانال پشتیبان سوئیچ می‌کند تا عملکرد مداوم را تضمین کند. این الگو ساده طراحی می‌شود و برای خطاهای تصادفی مؤثر است اما از خطاهای سیستماتیک محافظت نمی‌کند، زیرا هر خطای سیستماتیک در یک کانال در کپی‌های آن نیز وجود خواهد داشت. در حالی که این الگو قابلیت اطمینان بالایی در محیط‌های سخت ارائه می‌دهد، هزینه‌های بالاتری به دلیل نیاز به سخت‌افزار تکراری دارد.

۳.۱۰.۲ الگوی **Triple Modular Redundancy**

الگوی تکرار سه‌گانه مدولار (TMR) [۴] با استفاده از سه کانال موازی برای پردازش تسک‌ها، مقایسه خروجی‌ها و اعمال قانون دو از سه در صورت اختلاف، قابلیت اطمینان و ایمنی را افزایش می‌دهد. این الگو به سیستم اجازه می‌دهد تا در حضور خطاهای تصادفی بدون از دست دادن داده‌های ورودی یا نیاز به زمان اضافی برای تصحیح، به کار خود ادامه دهد. در حالی که هدف آن محافظت در برابر خطاهای تصادفی مشابه الگوی Homogeneous Redundancy است، عملیات موازی TMR آن را از نظر زمانی کارآمدتر می‌کند. با این حال، بدون استفاده از کانال‌های ناهمگن، از خطاهای سیستماتیک محافظت نمی‌کند. TMR به دلیل تکرار سخت‌افزار هزینه بالایی دارد، اما برای برنامه‌های بسیار حیاتی با نیاز به قابلیت اطمینان بالا و بدون وضعیت ایمن ضروری است.

۴.۱۰.۲ الگوی **Heterogeneous Redundancy**

الگوی تکرار ناهمگن [۴] با استفاده از چندین کانال که به‌طور مستقل طراحی شده‌اند، تشخیص خطاها را بهبود می‌بخشد و هم خطاهای سیستماتیک و هم خطاهای تصادفی را شناسایی می‌کند. این الگو به دلیل نیاز بیشتر به زحمت توسعه و هزینه‌های تکراری، پرهزینه‌تر از الگوی Homogeneous Redundancy است. برای سیستم‌های با ایمنی و قابلیت اطمینان بالا مناسب است و عملکرد مداوم را حتی در صورت بروز خطاها تضمین می‌کند. با این حال، هزینه‌های بالا به دلیل نیاز به طراحی‌های مستقل، معمولاً توسط تیم‌های مختلف به منظور جلوگیری از خطاهای سیستماتیک، ناشی می‌شود. این الگو حفاظت قوی در برابر خطاها ارائه می‌دهد اما با هزینه بیشتر، و به‌عنوان امن‌ترین اما پرهزینه‌ترین گزینه معماری در نظر گرفته می‌شود. ترکیب این الگو با الگوی Triple Modular Redundancy می‌تواند در دسترس بودن را بیشتر افزایش دهد.

۵.۱۰.۲ الگوی **Monitor-Actuator**

این الگو [۴] یک راهکار ایمنی مقرون‌به‌صرفه است که در سیستم‌هایی با نیازمندی‌های دسترسی متوسط تا پایین و حالت ایمن تعریف‌شده استفاده می‌شود. این الگو شامل یک حسگر مستقل است که کانال فعال‌سازی را برای شناسایی خطاها نظارت می‌کند و اطمینان می‌دهد که سیستم در صورت لزوم به حالت ایمن وارد می‌شود. این الگو شکل خاصی از الگوی Heterogeneous Redundancy است که به جای تکرار کامل کانال، نظارت را فراهم می‌کند. زمانی که سیستم می‌تواند با ورود به حالت ایمن، خطاها را تحمل کند، مناسب است و با کمترین تکرار اطمینان حاصل می‌شود که اگر یک کانال خراب شود، کانال دیگر می‌تواند خطا را شناسایی کرده یا به کار خود ادامه دهد.

۶.۱۰.۲ الگوی Sanity Check

این الگو [۴] در سیستم‌های نهفته بی‌درنگ یک روش سبک و کم‌هزینه برای اطمینان از عملکرد معقول سیستم، حتی اگر کاملاً دقیق نباشد، است. این الگو پوشش خطای حداقلی ارائه می‌دهد و برای شرایطی طراحی شده است که کنترل دقیق برای ایمنی حیاتی نیست، اما اقدامات نادرست می‌توانند ضرر رسان باشند. این الگو از حسگرهای ارزان‌قیمت و کم‌دقت برای شناسایی خطاهای قابل توجه در عملکرد استفاده می‌کند و مطمئن می‌شود که سیستم در صورت بروز انحرافات جزئی آسیب نمی‌بینند. این الگو یک نوع تغییر یافته از الگوی Monitor-Actuator است که به یک حالت ایمن در صورت بروز خطاهای بزرگ نیاز دارد و یک راه‌حل ساده و مقرون به صرفه برای محافظت حداقلی فراهم می‌کند.

۷.۱۰.۲ الگوی Watchdog

این الگو [۴] یک روش سبک و کم‌هزینه برای اطمینان از عملکرد صحیح فرآیندهای محاسبات داخلی است. برخلاف الگوی Sanity Check که خروجی سیستم را با استفاده از حسگرهای خارجی نظارت می‌کند، الگوی Watchdog بررسی می‌کند که محاسبات به درستی و به موقع انجام شوند. این الگو پوشش خطای حداقلی ارائه می‌دهد و عمدتاً خطاهای پایه زمانی و گیر افتادن احتمالی را شناسایی می‌کند. این الگو اغلب با الگوهای ایمنی دیگر ترکیب می‌شود تا قابلیت اطمینان سیستم را افزایش دهد، به‌ویژه در برنامه‌های حساس به زمان که محاسبات باید ضرب‌الاجل‌های دقیقی را رعایت کنند.

۸.۱۰.۲ الگوی Safety Executive

این الگو [۴] برای سیستم‌هایی طراحی شده است که اقدامات ایمنی پیچیده‌ای دارند و نمی‌توان آنها را به سادگی با خاموش کردن سیستم به دست آورد. این الگو یک جزء مجری ایمنی معرفی می‌کند تا چندین کانال و اقدامات ایمنی را مدیریت و هماهنگ کند و سیستم را از طریق یک سری مراحل به وضعیت ایمن هدایت کند. این الگو به‌ویژه برای سیستم‌هایی که با مواد خطرناک یا حالت‌های پرانرژی کار می‌کنند و خاموشی فوری می‌تواند خطرناک باشد، مفید است. پیاده‌سازی این الگو پیچیده است و معمولاً برای سیستم‌های بسیار حساس به ایمنی استفاده می‌شود و در چنین محیط‌هایی حفاظت عالی در برابر خطا ارائه می‌دهد.

۱۱.۲ الگوهای سخت‌افزاری برای سیستم‌های Safety-Critical

این دسته از الگوها که توسط Armoush در [۵] معرفی شده، شامل الگوهایی است که دارای تکرار در سخت‌افزار هستند و در حوزه امنیت و قابلیت اطمینان استفاده می‌شوند.

۱.۱۱.۲ الگوی Homogeneous Duplex

(این الگو همان الگوی Homogeneous Redundancy است که در [۴] گفته‌شده)

۲.۱۱.۲ الگوی Heterogeneous Duplex

(این الگو همان الگوی Heterogeneous Redundancy است که در [۴] نوشته‌شده.)

۳.۱۱.۲ الگوی Triple Modular Redundancy

(این الگو همان الگوی Triple Modular Redundancy است که در [۴] نوشته‌شده.)

۴.۱۱.۲ الگوی M-Out-Of-N

الگوی M out of N [۵] با ارائه تکرار همگن، شامل N ماژول یکسان است که به‌طور موازی برای مخفی کردن خطاهای تصادفی و افزایش ایمنی و قابلیت اطمینان سیستم کار می‌کنند. این الگو نیاز دارد که حداقل M از N ماژول برای عملکرد سیستم موفق شوند و از یک الگوریتم رأی‌گیری برای مدیریت خطاهای تصادفی بدون از دست دادن داده‌های ورودی استفاده می‌کند. این الگو برای سیستم‌هایی با نرخ بالای خطاهای تصادفی و بدون محدودیت تکرار مناسب است. تأثیر کمی بر قابلیت تغییرسیستم یا زمان اجرا دارد، اما در برابر خطاهای سیستماتیک بی‌اثر است، زیرا کانال‌های یکسانی دارند که دارای همان خطاهای احتمالی هستند. استفاده از چندین حسگر برای جلوگیری از خرابی نقطه‌ای واحد ممکن است به مشکلاتی منجر شود، به‌ویژه به دلیل تفاوت در زمان پاسخ حسگرها.

۵.۱۱.۲ الگوی Monitor-Actuator

(این الگو همان الگوی Monitor-Actuator است که در [۴] نوشته‌شده.)

۶.۱۱.۲ الگوی Sanity Check

(این الگو همان الگوی Sanity Check است که در [۴] نوشته‌شده.)

۷.۱۱.۲ الگوی Watchdog

(این الگو همان الگوی Watchdog است که در [۴] نوشته‌شده.)

۸.۱۱.۲ الگوی Safety Executive

(این الگو همان الگوی Safety Executive است که در [۴] نوشته‌شده.)

۱۲.۲ الگوهای نرم‌افزاری برای سیستم‌های Safety-Critical

این دسته از الگوها که در [۵] بیان شده‌اند، با استفاده از ساختارهای تکراری در نرم‌افزار می‌خواهند شرایط قابلیت اطمینان و امنیت را ایجاد کنند.

۱.۱۲.۲ الگوی N-Version Programming

الگوی N-Version Programming (NVP) [۵] یک روش نرم‌افزاری تحمل خطا است که بر تنوع نرم‌افزار و مخفی‌سازی خطاها متکی است. این روش شامل ایجاد N نسخه نرم‌افزاری معادل عملکردی ($N >= 2$) به‌صورت مستقل از مشخصات اولیه است. این نسخه‌ها به‌طور موازی اجرا می‌شوند و وظیفه یکسانی را با ورودی یکسان انجام می‌دهند تا N خروجی تولید کنند. در این الگو، یک رأی‌گیر برای تعیین خروجی صحیح با استفاده از نتایج N نسخه به کار می‌رود. NVP برای سیستم‌های با ایمنی بسیار بالا مناسب است، زمانی که نیاز به نرم‌افزار بسیار قابل اعتماد وجود دارد، هزینه بالای توسعه نسخه‌های متعدد قابل تحمل است، تیم‌های مستقل برای توسعه نسخه‌های مختلف موجود هستند و واحدهای سخت‌افزاری تکراری برای اجرای این نسخه‌ها به‌صورت موازی قابل استفاده هستند. با این حال، NVP دارای معایبی مانند پیچیدگی و هزینه بالای توسعه نسخه‌های مستقل و وابستگی به مشخصات اولیه است که می‌تواند خطاها را به تمامی نسخه‌ها منتقل کند و ایمنی و قابلیت اطمینان سیستم را تحت تأثیر قرار دهد.

۲.۱۲.۲ الگوی Recovery Block

این الگو [۵] یک روش نرم‌افزاری تحمل خطا است که از تشخیص خطا با آزمون‌های پذیرش و بازیابی اطلاعات برای جلوگیری از خرابی سیستم استفاده می‌کند. مشابه الگوی N-Version Programming، این روش شامل ایجاد N نسخه نرم‌افزاری متنوع، مستقل و معادل عملکردی از مشخصات اولیه است. این نسخه‌ها به نسخه اصلی و N-1 نسخه ثانویه تقسیم می‌شوند. اجرای هر نسخه با یک آزمون پذیرش دنبال می‌شود. اگر نسخه اصلی در آزمون خود شکست بخورد، یک نسخه ثانویه Recovery Block اجرا می‌شود و پس از آن آزمون پذیرش انجام می‌شود. این فرآیند تا زمانی که یک نسخه آزمون پذیرش را بگذارد یا همه نسخه‌ها شکست بخورند و یک خرابی کلی سیستم گزارش شود، تکرار می‌شود. این روش برای سیستم‌های با ایمنی بسیار بالا مناسب است، زمانی که نرم‌افزار بسیار قابل اعتماد و ایمنی نیاز است، امکان ساخت آزمون پذیرش برای اطمینان از عملکرد صحیح نرم‌افزار و تشخیص خروجی‌های ممکن نادرست وجود دارد، تیم‌های مستقل برای توسعه نسخه‌های مختلف موجود هستند و هزینه بالای توسعه نسخه‌های متعدد قابل تحمل است. معایب اصلی شامل وابستگی زیاد به کیفیت آزمون پذیرش، احتمال قطع سرویس در حین بازیابی و مشکلات مشترک با الگوی N-Version Programming مانند هزینه بالای توسعه، پیچیدگی توسعه نسخه‌های مستقل و وابستگی به مشخصات اولیه که ممکن است خطاهای نرم‌افزاری را به همه نسخه‌ها منتقل کند، می‌باشد.

۳.۱۲.۲ الگوی Acceptance Voting

الگوی رأی‌گیری پذیرش (AVP) [۵] یک روش نرم‌افزاری تحمل خطای ترکیبی است که عناصر الگوی N-Version Programming (NVP) و الگوی Recovery Block (RB) را ترکیب می‌کند. این روش شامل تولید مستقل N ($N >= 2$) ماژول نرم‌افزاری معادل از نظر عملکردی از مشخصات اولیه است. این نسخه‌ها به صورت موازی اجرا می‌شوند و همان وظیفه را بر روی همان ورودی انجام می‌دهند تا N خروجی تولید کنند. هر خروجی تحت آزمون پذیرش قرار می‌گیرد تا از صحت آن اطمینان حاصل شود. خروجی‌هایی که آزمون پذیرش را می‌گذرانند، به عنوان ورودی به یک رأی‌گیر پویا استفاده می‌شوند که خروجی صحیح را بر اساس یک طرح رأی‌گیری تعیین می‌کند. این الگو برای توسعه نرم‌افزار تحمل خطا برای سیستم‌های بسیار بحرانی از نظر ایمنی مناسب است، جایی که نیاز به نرم‌افزار بسیار قابل اعتماد است، آزمون‌های پذیرش قابل ساخت هستند، هزینه بالای پیاده‌سازی‌های متعدد قابل تحمل است، تیم‌های مستقل برای توسعه نسخه‌های مختلف

موجود هستند و امکان استفاده از واحدهای سخت‌افزاری اضافی برای اجرای نسخه‌ها به صورت موازی وجود دارد. مشابه روش اصلی که در الگوی N-Version Programming گفته‌شد، معایب اصلی AVP شامل تلاش برای توسعه نسخه‌های نرم‌افزاری متنوع و وابستگی زیاد به مشخصات اولیه است که ممکن است خطاها را به همه نسخه‌ها منتقل کند. با این حال، آزمون پذیرش یک اقدام اضافی برای تشخیص خطاهای وابسته فراهم می‌کند و مشکل خطاهای وابسته در این الگو کمتر بحرانی است.

۴.۱۲.۲ الگوی N-Self Checking Programming

این الگو [۵] یک روش نرم‌افزاری تحمل خطای بسیار پرهزینه است که بر تنوع طراحی نرم‌افزار و خود بررسی از طریق تکرار تأکید دارد. این روش شامل تولید مستقل حداقل چهار ماژول نرم‌افزاری معادل عملکردی از مشخصات اولیه است. این نسخه‌ها به گروه‌هایی به نام اجزا مرتب می‌شوند، که هر جزء شامل دو نسخه و یک الگوریتم مقایسه برای بررسی صحت نتایج است. در طول اجرا، یک جزء به طور فعال خدمت مورد نیاز را ارائه می‌دهد، در حالی که اجزای دیگر به عنوان یدک‌های آماده عمل می‌کنند. برای اطمینان از تحمل خطا برای یک خطا، حداقل چهار نسخه باید بر روی چهار واحد سخت‌افزاری اجرا شوند، که آن را به پرهزینه‌ترین روش در مقایسه با سایر روش‌ها تبدیل می‌کند. این الگو برای توسعه نرم‌افزار تحمل خطا برای سیستم‌های بسیار بحرانی از نظر ایمنی مناسب است، جایی که نیاز به نرم‌افزار بسیار قابل اعتماد است، هزینه بالای پیاده‌سازی‌های متعدد قابل تحمل است، تیم‌های مستقل برای توسعه نسخه‌های مختلف موجود هستند و امکان استفاده از واحدهای سخت‌افزاری اضافی برای اجرای این نسخه‌ها به صورت موازی وجود دارد. معایب اصلی این الگو شامل وابستگی زیاد به مشخصات اولیه که ممکن است خطاها را به همه نسخه‌ها منتقل کند، تعداد بالای نسخه‌های متنوع و ماژول‌های سخت‌افزاری مورد استفاده در مقایسه با سایر الگوها که همان تعداد خطا را تحمل می‌کنند، و پیچیدگی توسعه N نسخه مستقل و معادل عملکردی است.

۵.۱۲.۲ الگوی Recovery Block with Backup Voting

این الگو [۵] یک الگوی ترکیبی است که ایده‌های الگوی Recovery Block را با الگوی N-Version Programming ترکیب می‌کند تا قابلیت اطمینان را به ویژه در مواقعی که ساخت آزمون پذیرش مؤثر دشوار است، بهبود بخشد. این الگو از N ماژول نرم‌افزاری مستقل و متنوع و معادل از نظر عملکردی که از همان مشخصات اولیه توسعه یافته‌اند، استفاده می‌کند. این الگو به مشکل موارد منفی کاذب در آزمون‌های پذیرش که در آن خروجی‌های صحیح به اشتباه به عنوان خروجی‌های نادرست شناسایی می‌شوند، می‌پردازد. در این الگو، نسخه اصلی ابتدا اجرا می‌شود و سپس یک آزمون پذیرش انجام می‌شود. اگر نسخه اصلی موفق نشود، خروجی آن در حافظه کش به عنوان پشتیبان ذخیره می‌شود و نسخه جایگزین بعدی برای انجام عملکرد مورد نیاز فراخوانی می‌شود. این فرآیند تا زمانی ادامه می‌یابد که یک نسخه جایگزین آزمون پذیرش را بگذراند. اگر همه نسخه‌های جایگزین شکست بخورند، خروجی‌های ذخیره شده به عنوان ورودی به یک روش رأی‌گیری استفاده می‌شوند تا نتیجه معتبری تعیین شود. این الگو برای توسعه نرم‌افزار برای سیستم‌های بسیار بحرانی از نظر ایمنی مناسب است که نیاز به نرم‌افزار بسیار قابل اعتماد دارند، ساخت یک آزمون پذیرش مؤثر دشوار است، تیم‌های مستقل برای توسعه نسخه‌های مختلف در دسترس هستند و هزینه بالای پیاده‌سازی‌های متعدد قابل تحمل است. هدف این الگو بهبود قابلیت اطمینان نرم‌افزار و رفع مشکل موارد منفی کاذب در آزمون‌های پذیرش ضعیف است. معایب اصلی این الگو مشابه الگوی Recovery Block است، از جمله پیچیدگی توسعه N نسخه مستقل، هزینه‌های بالای توسعه و وابستگی زیاد به مشخصات اولیه که ممکن است خطاها را به تمام نسخه‌ها منتقل کند.

۱۳.۲ الگوهای ترکیبی سخت‌افزار و نرم‌افزار برای سیستم‌های Safety-Critical

این دسته از الگوها که در [۵] آمده‌است، به صورت مشخص درباره تکرار هیچ‌کدام از سخت‌افزار یا نرم‌افزار صحبت نمی‌کند.

۱.۱۳.۲ الگوی Protected Single Channel

این الگو همان الگوی Protected Single Channel است که در [۴] آمده‌است.

۲.۱۳.۲ الگوی 3-Level Safety Monitoring

این الگو [۵] یک ترکیب از الگوی Monitor-Actuator و الگوی Watchdog است که برای برنامه‌هایی مناسب است که نیاز به مانیتورینگ ایمنی مداوم دارند و شامل یک حالت ایمن در صورت خرابی هستند، بدون اینکه به سخت‌افزارهای تکراری زیادی نیاز داشته باشند. این الگو شامل یک کانال سخت‌افزاری واحد است که به سه سطح تقسیم می‌شود: سطح عملکرد، مانیتورینگ و کنترل. سطح عملکرد زیربرنامه‌ای را برای انجام عملکرد مورد نظر اجرا می‌کند، سطح مانیتورینگ سطح عملکرد را نظارت می‌کند و سطح کنترل سطح مانیتورینگ و کل کانال سخت‌افزاری را کنترل می‌کند. علاوه بر این، یک Watchdog که از طریق پیام‌های دوره‌ای با سطح کنترل ارتباط برقرار می‌کند، برای بازنشانی سیستم به حالت ایمن در صورت خرابی استفاده می‌شود.

این الگو برای توسعه سیستم‌های نهفته با یک حالت ایمن یا اقدام اصلاحی مشخص و بدون تکرار سخت‌افزاری مناسب است. هدف آن بهبود ایمنی سیستم تعبیه شده با هزینه معقول در حضور خرابی در سیستمی با حالت ایمن است. علاوه بر این، چگونه می‌توان سطح ایمنی مورد نیاز را حفظ کرد و اطمینان حاصل کرد که سیستم آسیبی وارد نمی‌کند، زمانی که انحرافی در خروجی Acutatorها از نقطه تنظیم فرمان وجود دارد. عیب اصلی این الگو این است که شامل یک کانال سخت‌افزاری واحد است؛ بنابراین، نمی‌توان از آن برای تحمل خرابی‌های سخت‌افزاری در برنامه‌هایی با نیازهای بالای قابلیت اطمینان و دسترسی استفاده کرد.

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.