



دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

عنوان:

الگوها در سیستم های نهفته بی درنگ

نویسنده

علی محسنی نژاد

استاد

دکتر رامان رامسین

مرداد ۱۴۰۳

۶	الگوهای طراحی برای دسترسی به سخت افزار	۱.۲
۶	الگوی Hardware Proxy	۱.۱.۲
۷	الگوی Hardware Adapter	۲.۱.۲
۸	الگوی Mediator	۳.۱.۲
۸	الگوی Observer	۴.۱.۲
۹	الگوی Debouncing	۵.۱.۲
۹	الگوی Interrupt	۶.۱.۲
۱۱	الگوی Polling	۷.۱.۲
۱۲	الگوهای طراحی برای همزمانی نهفته و مدیریت حافظه	۲.۲
۱۲	الگوی Cyclic Executive	۱.۲.۲
۱۳	الگوی Static Priority	۲.۲.۲
۱۳	الگوی Critical Region	۳.۲.۲
۱۵	الگوی Guarded Call	۴.۲.۲
۱۵	الگوی Queuing	۵.۲.۲
۱۶	الگوی Rendezvous	۶.۲.۲
۱۷	الگوی Simultaneous Locking	۷.۲.۲
۱۸	الگوی Ordered Locking	۸.۲.۲
۱۹	الگوهای طراحی برای ماشین های حالت	۳.۲
۱۹	الگوی Single Event Receptor	۱.۳.۲
۱۹	الگوی Multiple Event Receptor	۲.۳.۲
۱۹	الگوی State Table	۳.۳.۲
۱۹	الگوی State	۴.۳.۲
۱۹	الگوی Decomposed And State	۵.۳.۲
۱۹	الگوهای امنیت و قابلیت اطمینان	۴.۲
۱۹	الگوی One's Complement	۱.۴.۲
۲۰	الگوی CRC	۲.۴.۲
۲۱	الگوی Smart Data	۳.۴.۲
۲۲	الگوی Channel	۴.۴.۲
۲۲	الگوی Protected Single Channel	۵.۴.۲
۲۲	الگوی Dual Channel	۶.۴.۲
۲۲	الگوهای معماری زیربخش ها و اجزا	۵.۲
۲۳	الگوی Layered	۱.۵.۲
۲۳	الگوی Five Layer	۲.۵.۲
۲۳	الگوی Microkernel	۳.۵.۲
۲۴	الگوی Channel	۴.۵.۲
۲۶	الگوی Recursive Containment	۵.۵.۲
۲۶	الگوی Hierarchical Control	۶.۵.۲
۲۶	الگوی Virtual Machine	۷.۵.۲
۲۷	معماری Component-Based	۸.۵.۲
۲۹	الگوی ROOM	۹.۵.۲
۲۹	الگوهای معماری همزمانی	۶.۲
۲۹	الگوی Message Queuing	۱.۶.۲

۲۹	Interrupt	الگوی	۲.۶.۲	
۳۰	Guarded Call	الگوی	۳.۶.۲	
۳۱	Rendezvous	الگوی	۴.۶.۲	
۳۱	Cyclic Execution	الگوی	۵.۶.۲	
۳۱	Round Robin	الگوی	۶.۶.۲	
۳۲	Static Priority	الگوی	۷.۶.۲	
۳۲	Dynamic Priority	الگوی	۸.۶.۲	
۳۳	معماری حافظه	الگوهای	۷.۲	
۳۳	Static Allocation	الگوی	۱.۷.۲	
۳۳	Pool Allocation	الگوی	۲.۷.۲	
۳۴	Fixed Sized Buffer	الگوی	۳.۷.۲	
۳۵	Smart Pointer	الگوی	۴.۷.۲	
۳۵	Garbage Collection	الگوی	۵.۷.۲	
۳۶	Garbage Compactor	الگوی	۶.۷.۲	
۳۶	معماری منابع	الگوهای	۸.۲	
۳۶	Critical Section	الگوی	۱.۸.۲	
۳۶	Priority Inheritance	الگوی	۲.۸.۲	
۳۷	Highest Locker	الگوی	۳.۸.۲	
۳۸	Priority Ceiling	الگوی	۴.۸.۲	
۳۸	Simultaneous Locking	الگوی	۵.۸.۲	
۳۸	Ordered Locking	الگوی	۶.۸.۲	
۳۸	معماری توزیع	الگوهای	۹.۲	
۳۹	Shared Memory	الگوی	۱.۹.۲	
۳۹	Remote Method Call	الگوی	۲.۹.۲	
۴۰	Observer	الگوی	۳.۹.۲	
۴۰	Data Bus	الگوی	۴.۹.۲	
۴۱	Proxy	الگوی	۵.۹.۲	
۴۲	Broker	الگوی	۶.۹.۲	
۴۲	معماری امنیت و قابلیت اطمینان	الگوهای	۱۰.۲	
۴۲	Protected Single Channel	الگوی	۱.۱۰.۲	
۴۵	Homogeneous Redundancy	الگوی	۲.۱۰.۲	
۴۵	Triple Modular Redundancy	الگوی	۳.۱۰.۲	
۴۷	Heterogeneous Redundancy	الگوی	۴.۱۰.۲	
۴۸	Monitor-Actuator	الگوی	۵.۱۰.۲	
۴۹	Sanity Check	الگوی	۶.۱۰.۲	
۴۹	Watchdog	الگوی	۷.۱۰.۲	
۴۹	Safety Executive	الگوی	۸.۱۰.۲	
۴۹	Safety-Critical	الگوهای سخت‌افزاری برای سیستم‌های	۱۱.۲	
۵۰	Homogeneous Duplex	الگوی	۱.۱۱.۲	
۵۰	Heterogeneous Duplex	الگوی	۲.۱۱.۲	
۵۰	Triple Modular Redundancy	الگوی	۳.۱۱.۲	
۵۱	M-Out-Of-N	الگوی	۴.۱۱.۲	
۵۱	Monitor-Actuator	الگوی	۵.۱۱.۲	
۵۱	Sanity Check	الگوی	۶.۱۱.۲	
۵۱	Watchdog	الگوی	۷.۱۱.۲	
۵۱	Safety Executive	الگوی	۸.۱۱.۲	
۵۱	Safety-Critical	الگوهای نرم‌افزاری برای سیستم‌های	۱۲.۲	

۵۱	N-Version Programming	الگوی	۱.۱۲.۲
۵۱	Recovery Block	الگوی	۲.۱۲.۲
۵۲	Acceptance Voting	الگوی	۳.۱۲.۲
۵۲	N-Self Checking Programming	الگوی	۴.۱۲.۲
۵۲	Recovery Block with Backup Voting	الگوی	۵.۱۲.۲
۵۲	Safety-Critical	الگوهای ترکیبی سخت افزار و نرم افزار برای سیستم های	۱۳.۲
۵۳	Protected Single Channel	الگوی	۱.۱۳.۲
۵۳	3-Level Safety Monitoring	الگوی	۲.۱۳.۲
۵۳	فریم ورک ساخت الگوهای معماری		۱۴.۲
۵۴		تحلیل	۳
۵۴	الگوهای معماری		۱.۳
۵۴	From Mud to Structure		۱.۱.۳
۵۴	Distributed Systems		۲.۱.۳
۵۵	Interactive Systems		۳.۱.۳
۵۵	Adaptable Systems		۴.۱.۳
۵۵	Safe Channel-based Systems		۵.۱.۳
۵۵	Safe Systems without Channel		۶.۱.۳
۵۶	الگوهای طراحی		۲.۳
۵۶	Hardware Patterns		۱.۲.۳
۵۶	Concurrent Scheduling Patterns		۲.۲.۳
۵۷	Concurrent Resource Sharing Patterns		۳.۲.۳
۵۷	State Machine Patterns		۴.۲.۳
۵۸	Safety with Redundancy Patterns		۵.۲.۳
۵۸	Safety without Redundancy Patterns		۶.۲.۳
۵۸	Memory Patterns		۷.۲.۳
۵۹		مراجع	۴

۱ مقدمه

این گزارش به طور مفصل به توضیح الگوهای معرفی شده 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 داده شده 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 صرفاً 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 ارائه شده در این بخش به صورت خطی، کتاب‌ها و مقالات بیان شده را بررسی کرده و الگوهای بیان شده از طرف ایشان را با همان ساختار و دسته‌بندی مورد نظر آن منبع ذکر کرده‌است. در فصل ۳ به بررسی و دسته‌بندی الگوهای گفته شده در فصل ۲ می‌پردازیم. در این بخش علاوه بر معرفی یک دسته‌بندی از الگوها، ارتباط بین الگوها نیز بیان می‌شود.

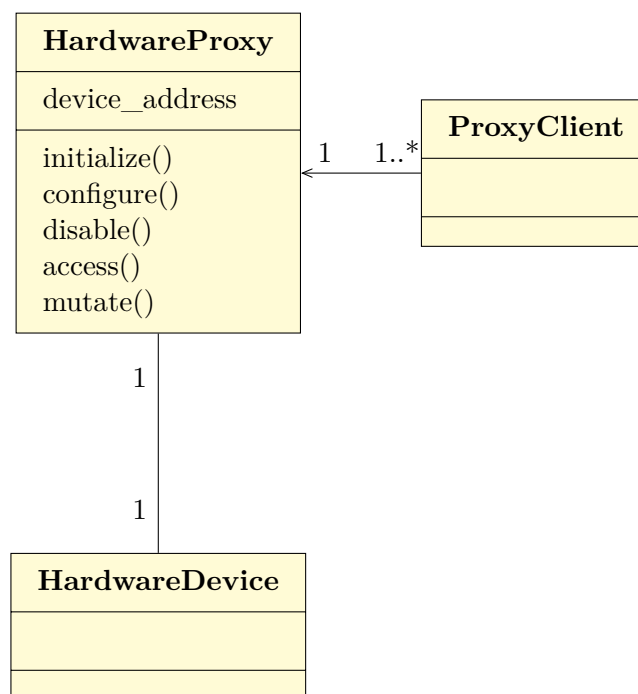
۲ پیشینه پژوهش

۱.۲ الگوهای طراحی برای دسترسی به سخت افزار

نرم افزارهای نهفته بر روی یک بستر سخت افزاری مستقر می شوند و معمولاً بسیاری از قابلیت های آن ها ملزم به ارتباط با سخت افزار می شود. به همین دلیل Douglass [۱] یک دسته از الگوها را با عنوان الگوهای دسترسی به سخت افزار معرفی می کند.

۱.۱.۲ الگوی Hardware Proxy

این الگو [۱] با ایجاد یک رابط روی یک جزء سخت افزاری، یک دسترسی مستقل از پیچیدگی های اتصال به سخت افزار برای کلاینت ایجاد می کند. این الگو با معرفی یک کلاس به نام پروکسی بین سخت افزار و کلاینت، باعث می شود که تمامی عملیات وابسته به سخت افزار در پروکسی انجام شود و در صورت تغییر در سخت افزار، هیچ تغییری به کلاینت تحمیل نشود. در این الگو بر روی یک جزء سخت افزاری، یک پروکسی قرار گرفته و کلاینت های متعدد می توانند از آن سرویس بگیرند. لازم به ذکر است که ارتباط پروکسی و سخت افزار بر پایه یک «رابط قابل آدرس دهی توسط نرم افزار» است. دیاگرام کلاس این الگو در شکل ۱ رسم شده است.



شکل ۱: دیاگرام کلاس Hardware Proxy

همانطور که در شکل ۱ دیده می شود، کلاس پروکسی توابع مشخصی را در اختیار کلاینت ها قرار می دهد^۱. توضیحات مربوط به هر یک از توابع کلاس پروکسی در شکل زیر داده شده است:

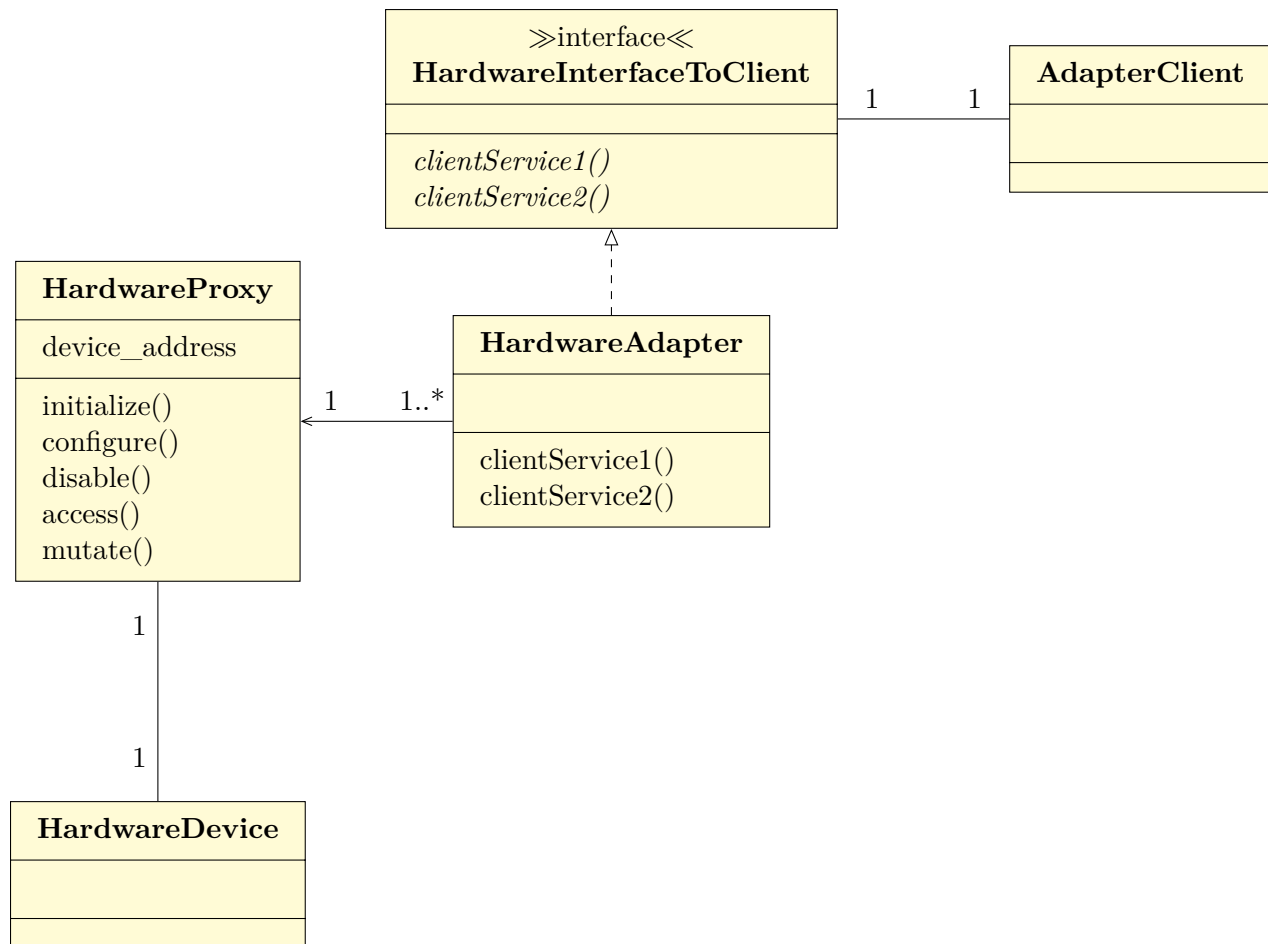
- initialize: این تابع برای آماده سازی اولیه ارتباط با سخت افزار استفاده می شود و معمولاً تنها یک بار صدا زده می شود.
- configure: این تابع برای ارسال تنظیمات برای سخت افزار استفاده می شود. معمولاً باید در سخت افزار تنظیماتی قرار داده شود که آن را قابل استفاده کند.
- disable: این تابع برای غیرفعال کردن سخت افزار به صورت امن استفاده می شود.
- access: این تابع برای دریافت اطلاعات از طرف سخت افزار استفاده می شود.
- mutate: این تابع برای فرستادن اطلاعات به سمت سخت افزار استفاده می شود.

^۱توابع دیگری نیز در [۱] گفته شده ولی اینجا تنها توابع public کلاس پروکسی را بررسی می کنیم.

این الگو بسیار رایج است و مزایای کپسوله سازی رابط سخت افزار و جزئیات کدگذاری را فراهم می کند، به طوری که تغییرات رابط سخت افزار بدون نیاز به تغییر در کلاینت ها انجام می شود. این کپسوله سازی می تواند تأثیر منفی بر عملکرد زمان اجرا داشته باشد، زیرا کاربران از فرمت اصلی داده ها آگاه نیستند. با این حال، آگاهی کلاینت ها از جزئیات کدگذاری باعث پیچیدگی در نگهداشت سیستم می شود.

۲.۱.۲ الگوی Hardware Adapter

این الگو [۱] مشابه الگوی Adapter که Gamma و دیگران [۳] معرفی کرده اند تعریف شده. استفاده از این الگو این اجازه را می دهد که کلاینتی که انتظار یک رابط خاص با سخت افزار را دارد، بتواند با سخت افزارهای مختلف بدون این که متوجه تفاوت های آن ها شود ارتباط بگیرد. این الگو روی ساختار الگوی Hardware Proxy بنا شده است و دیاگرام کلاس آن در شکل ۲ ترسیم شده است.



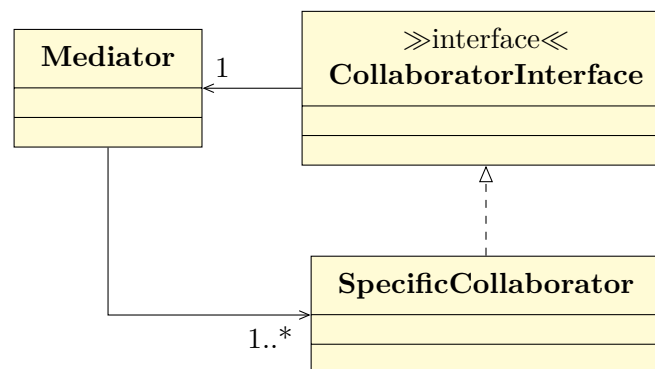
شکل ۲: دیاگرام کلاس Hardware Adapter

همانطور که در شکل ۲ دیده می شود، کلاس کلاینت سرویس های مورد انتظار خود را از رابط **HardwareInterfaceToClient** انتظار دارد. در این ساختار، کلاس آداپتور، سرویس های مورد انتظار کلاینت را به سرویس های ارائه شده از طرف سخت افزار ترجمه می کند. این کار اجازه می دهد که در صورت تغییر سخت افزار (و متناظرا پروکسی)، تنها با ایجاد پیاده سازی جدید برای رابط آداپتور، نیازی به تغییر در کلاینت نباشد.

استفاده از این الگو اجازه می دهد پروکسی های سخت افزار و دستگاه های مرتبط در برنامه های مختلف بدون تغییر استفاده شوند و برنامه های موجود نیز بدون تغییر از دستگاه های سخت افزاری مختلف استفاده کنند. **Adapter** به عنوان پل ارتباطی بین پروکسی سخت افزار و برنامه عمل می کند، که تغییر یا استفاده مجدد از دستگاه های سخت افزاری را آسان تر، سریع تر و کم خطا تر می سازد. 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 میانجی گر بین چند کلاس همکار، کمک می کند که چند سخت افزار را با هم مدیریت کند. ساختار این الگو در شکل ۳ ترسیم شده است.



شکل ۳: دیاگرام کلاس Mediator

همانطور که در شکل مشخص است، کلاس میانجی با هر یک از کلاس های همکار ارتباط دارد. این ارتباط به این شکل است که کلاس میانجی تمامی پیاده سازی های رابط همکار را می شناسد و با آن ها ارتباط دارد. این کلاس ها خودشان نیز همانطور که نشان داده شده، میانجی را می شناسند و با آن ارتباط دارند. هر یک از کلاس های همکار، با سخت افزار در ارتباط هستند و حتی می توانند 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 سخت افزار و نیاز است، توسط میانجی کنترل می شود.

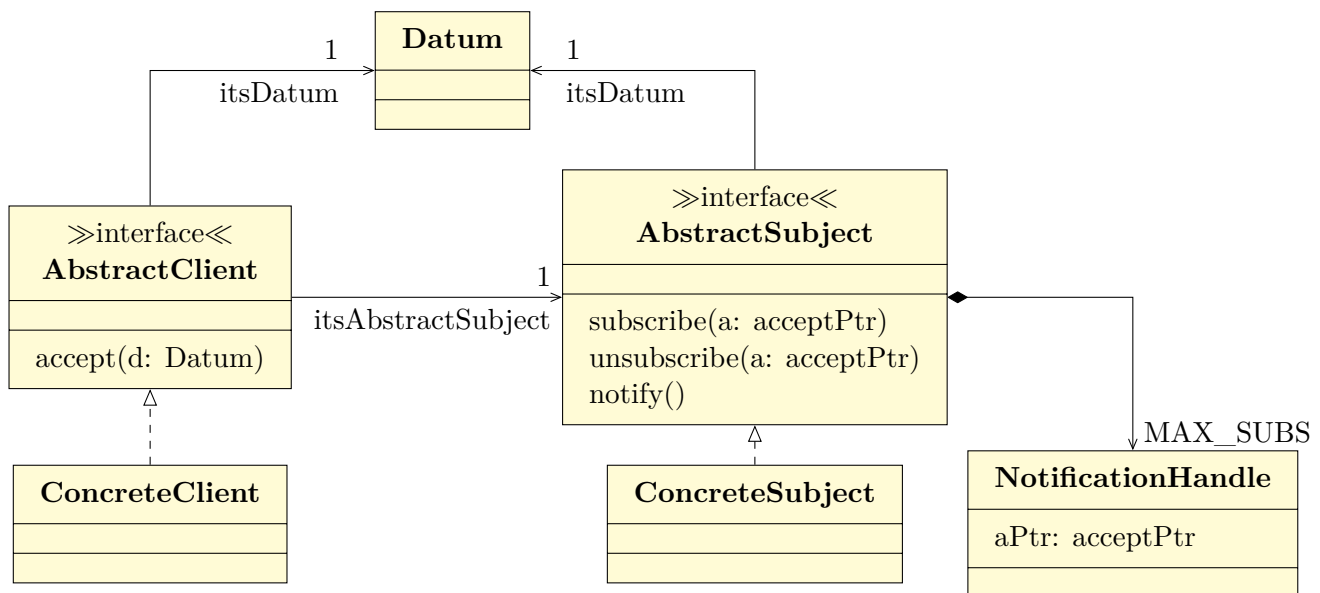
این الگو یک میانجی ایجاد می کند که هماهنگی بین مجموعه ای از عملگرهای همکار را بدون نیاز به اتصال مستقیم آن ها انجام می دهد، که طراحی کلی را ساده تر می کند. میانجی به جای تماس مستقیم همکاران با یکدیگر، اطلاعات را دریافت و به صورت کلی پاسخ می دهد. با توجه به نیاز بسیاری از سیستم های نهفته به واکنش سریع، تأخیرها می توانند 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 سیستم های نهفته، الگوی Observer [۱] است. این الگو به شیء های برنامه این اجازه را می دهد 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 داده را برای سرور بفرستد، برای آن سرور درخواست عضویت فرستاده و سرور هرگاه که داده های جدید در دسترس بودند، آن ها را برای کلاینت های عضو شده بفرستد. یکی از مهم ترین کاربردهای این الگو در دریافت داده ها از سنسورها است. یکی از قابلیت های خوب این الگو این است که کلاینت ها می توانند در زمان اجرای برنامه عضویت خود را قطع یا ایجاد کنند. در شکل ۴ دیاگرام کلاس این الگو را می بینیم.

در این ساختار کلاینت ها با فرستادن یک اشاره گر به کلاس سابجکت، درخواست عضویت برای سرویس می فرستند. کلاس سابجکت نیز با ذخیره کردن اشاره گرهای مختلف از طرف کلاینت ها زمانی که داده جدید آماده می شود، با فراخوانی تابع notify، تابع accept که اشاره گرهایش را در NotificationHandle قرار داده، با پاس دادن ورودی به فرمت Datum صدا می زند. اینگونه این داده برای تمامی کلاینت های عضو سرویس فرستاده می شود. کلاینت ها می توانند در حین اجرای برنامه، عضویت خود برای سرویس را لغو کنند. دقت شود که خود کلاس های سابجکت معمولاً از نوع پروکسی هستند (الگوی Hardware Proxy).

این الگو فرآیند توزیع داده ها به مجموعه ای از کلاینت ها را که ممکن است در زمان طراحی مشخص نباشند، ساده می کند و مدیریت داینامیک لیست کلاینت های علاقه مند در زمان اجرا را فراهم می کند. این الگو رابطه اصلی کلاینت-سرور را حفظ می کند و از طریق مکانیزم اشتراک گذاری، انعطاف پذیری در زمان اجرا ارائه می دهد. بهره وری محاسباتی حفظ می شود زیرا کلاینت ها تنها زمانی که مناسب است به روزرسانی می شوند؛ سیاست رایج این است که مشتریان در زمان تغییر داده ها به روزرسانی شوند، اما هر سیاست مناسب دیگری نیز می تواند اعمال شود.



شکل ۴: دیاگرام کلاس Observer

۵.۱.۲ الگوی Debouncing

در سخت‌افزار بسیاری از ورودی‌ها به صورت دکمه‌ها و سویچ‌هایی هستند که بر اثر ایجاد اتصال دو فلز با یکدیگر، باعث فعال شدن 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 میلی ثانیه 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.

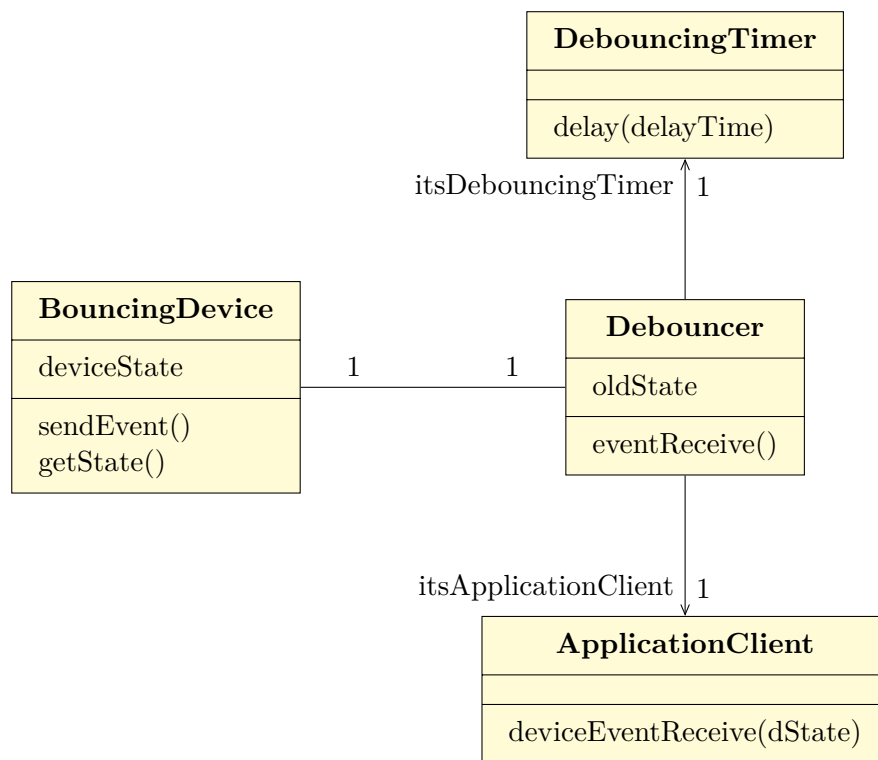
این الگو [۱] به ما کمک می‌کند که با صبر کردن برای یک مدت کوتاه، مقدار ورودی را زمانی که پایدار شده‌است بخوانیم. با این کار دغدغه معتبر بودن مقدار خوانده‌شده را در کلاینت نخواهیم داشت. دیاگرام کلاسی این الگو در شکل ۵ نمایش داده شده‌است. در این ساختار، BouncingDevice همان سخت‌افزار مورد بررسی است. تابع sendEvent می‌تواند یک نوع اینترپیت 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 پایدار شدن شرایط خروجی سخت‌افزار، خواندن حالت سخت‌افزار را به تعویق می‌اندازد.

۶.۱.۲ الگوی Interrupt

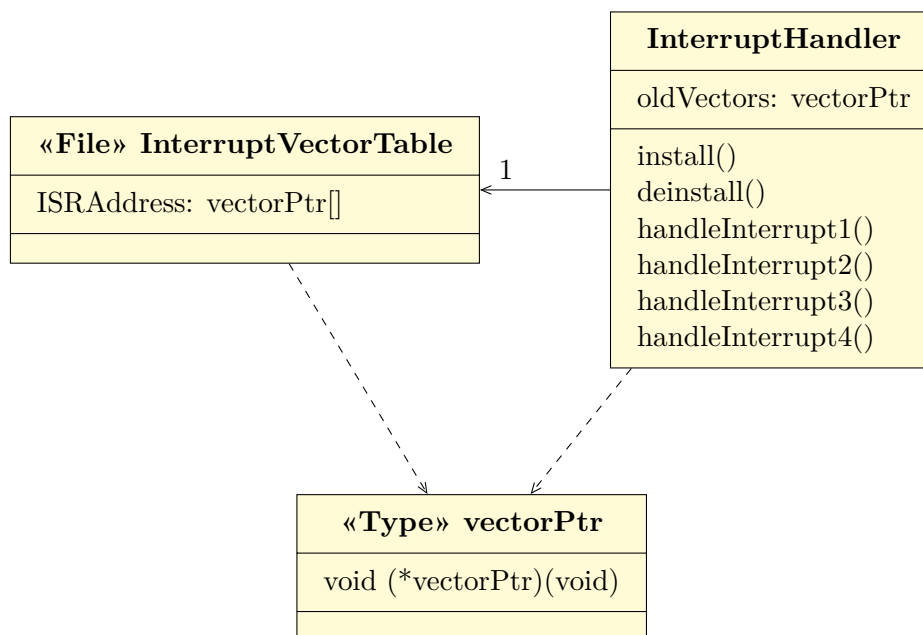
یکی از واحدهای مهم در سیستم‌های سخت‌افزاری، واحد Interrupt است. Interrupt برای هندل کردن وقایعی است که توسط سخت‌افزار جرقه زده می‌شوند. زمانی که یک interrupt رخ می‌دهد، نرم‌افزار فرایند کاری اصلی خود را متوقف کرده و یک فرایند رسیدگی به interrupt اتفاق افتاده آغاز می‌شود. با انجام این فرایند و رسیدگی 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 جایگزین کرد. 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 صرفاً یک نوع اشاره‌گر به تابعی 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

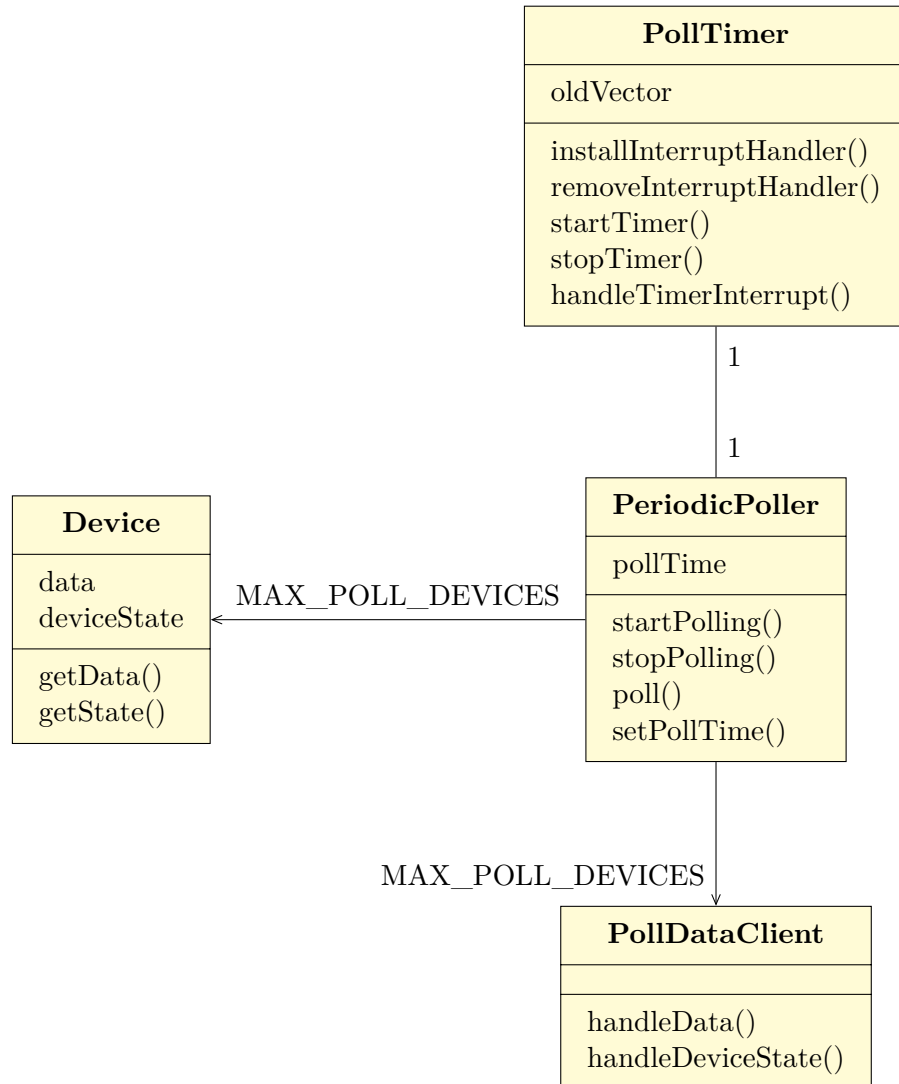


شکل ۶: دیاگرام کلاس Interrupt

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

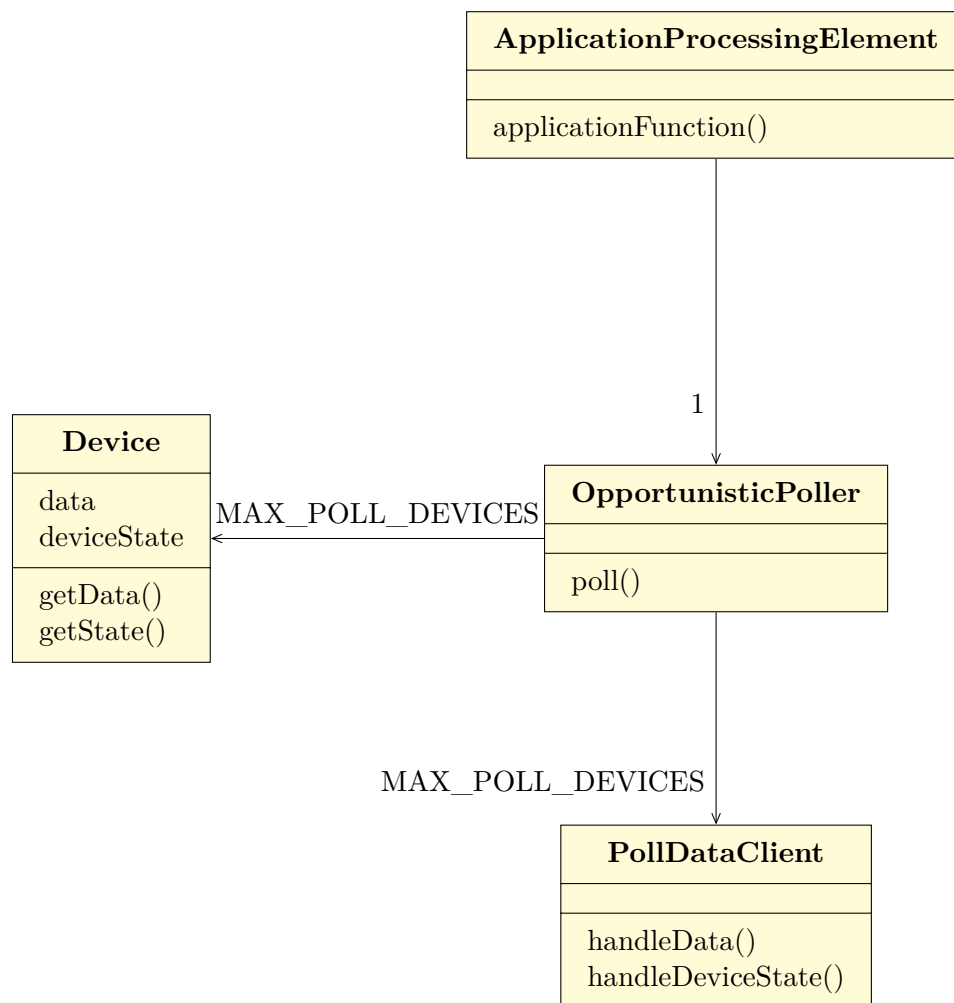
این الگو [۱] یک روش دیگر برای دریافت داده‌ها از سنسورها است و زمانی استفاده می‌شود که استفاده از الگوی Interrupt ممکن نیست یا این که داده‌هایی که می‌خواهیم ضبط کنیم آن قدر اضطراری نیستند و می‌توان برای دریافت آن‌ها صبر کرد. عملکرد این الگو به این صورت است که با سرکشی کردن به صورت دوره‌ای داده‌ها را دریافت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

`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

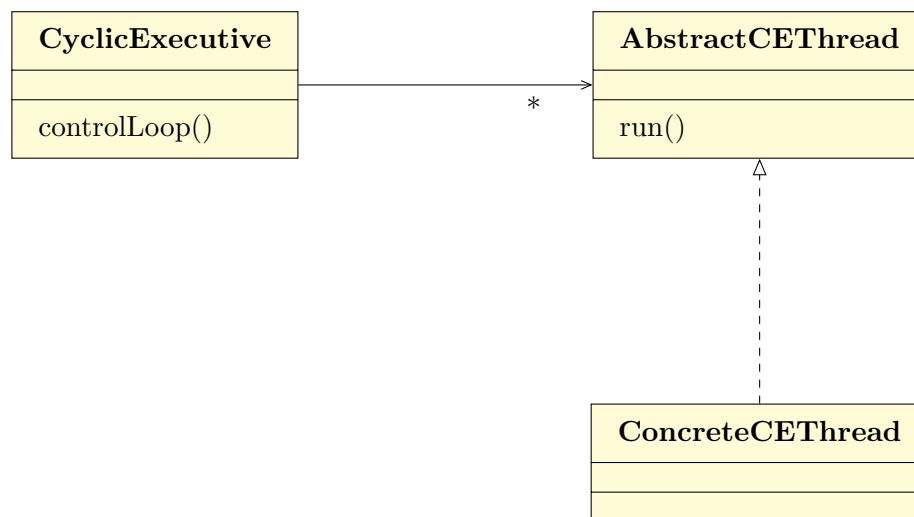
در بسیاری از مواقع در سیستم‌های نهفته لازم است که فعالیت‌های متنوع به صورت همزمان انجام شوند. به همین دلیل Douglass [۱] یک دسته از الگوها به نام الگوهای برنامه‌ریزی را معرفی کرده‌است.

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

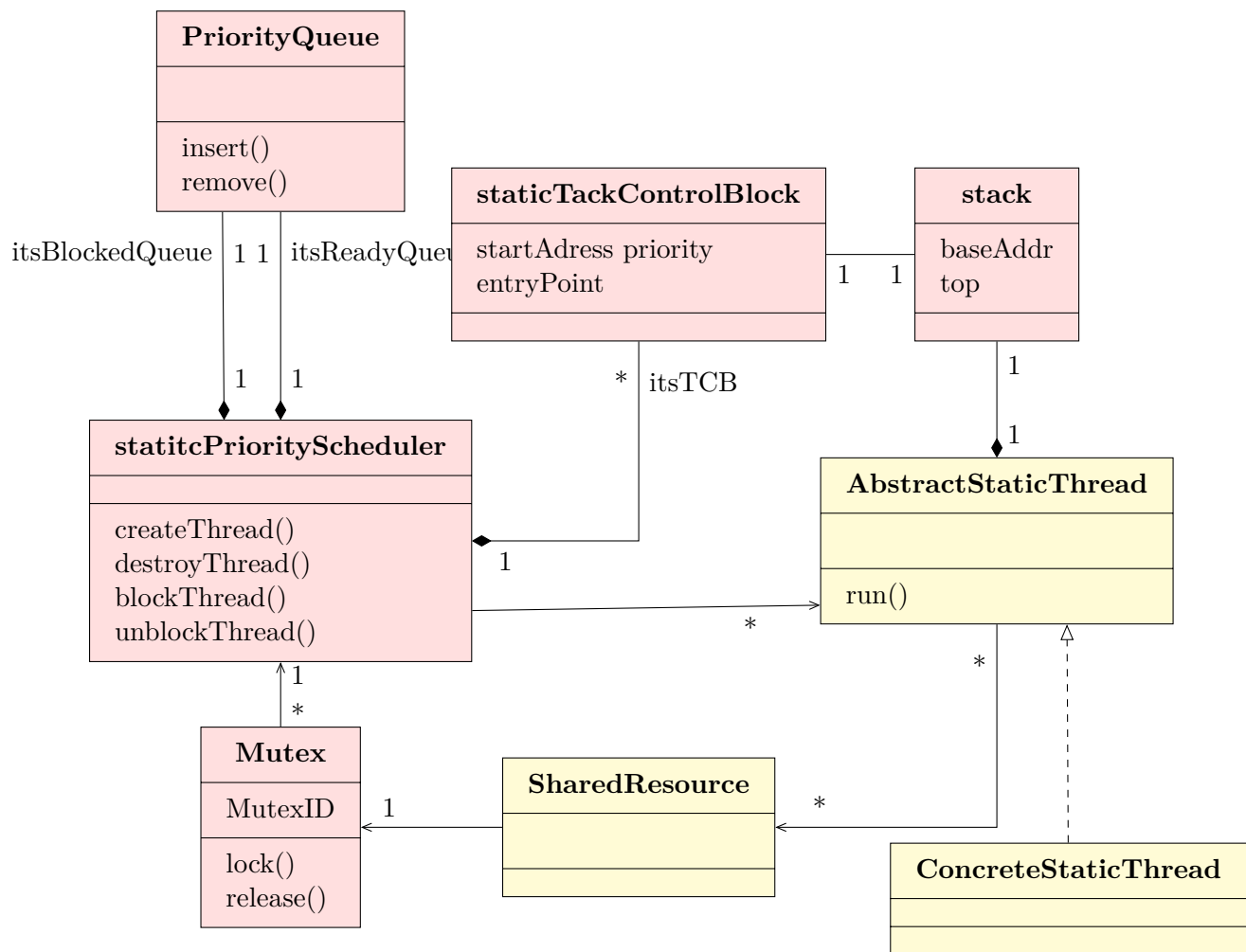
این [۱] یکی از پرکاربردترین الگوهای برنامه‌ریزی در سیستم‌های نهفته بی‌درنگ است. این الگو به ما این قدرت را می‌دهد تا بتوانیم با استفاده از یک سیستم اولویت‌دهی به تسک‌ها، آن‌ها را انجام دهیم. در این سیستم فرض می‌شود که همه تسک‌ها از نوع سنکرون هستند و آن‌ها را بر اساس زمان ددلاینشان اولویت‌دهی می‌کنیم.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

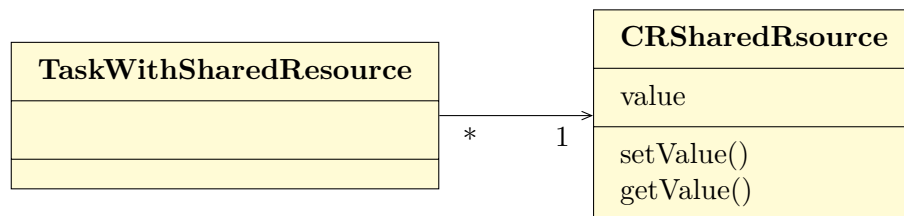
۳.۲.۲ الگوی Critical Region

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 سوییچ کردن بین تسک‌ها را متوقف می‌کنیم تا زمانی که این تسک به پایان برسد. سپس دوباره فرایند زمان‌بندی تسک‌ها و سوییچ کردن بین آن‌ها به حالت عادی بازمی‌گردد. استفاده از این الگو معمولاً 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 سوییچ کردن 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 سوییچ کردن بین تسک‌ها را در زمان انجام این تسک متوقف کنیم.



شکل ۱۰: دیگرام 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 سوییچ کردن 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 سوییچ کردن تسک‌ها را شروع کند در صورتی که تابع فراخوان هم‌چنان در حالت بحرانی است.



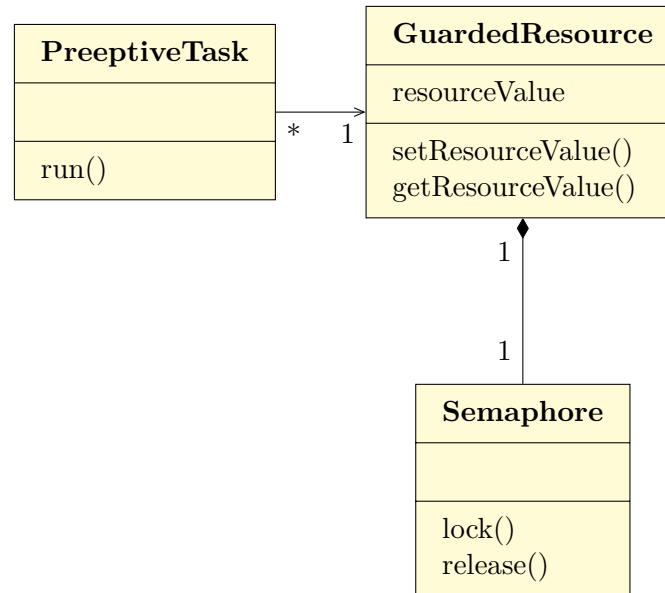
شکل ۱۱: دیگرام کلاس Critical Region

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** هندل شده.^۳

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

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 به موقع 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 به موقع 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 پشتیبانی شده توسط RTOS می‌توانند این مشکل را کاهش دهند. ساختار کلاسی این الگو در شکل ۱۲ نمایش داده شده است.



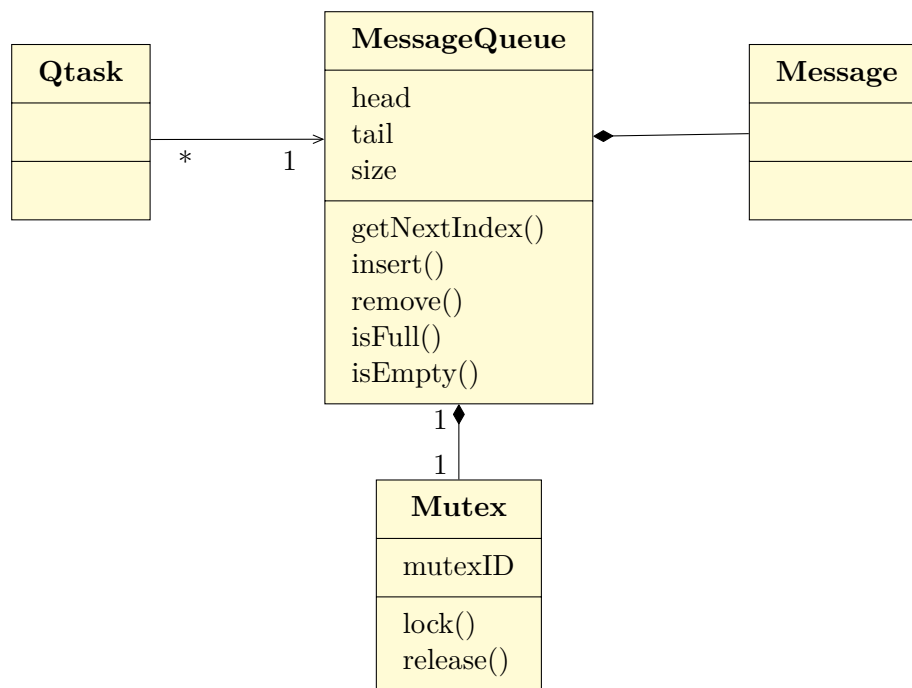
شکل ۱۲: دیاگرام کلاس Guarded Call

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 صرفاً هدف نشان دادن 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 آلوده شدن 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.

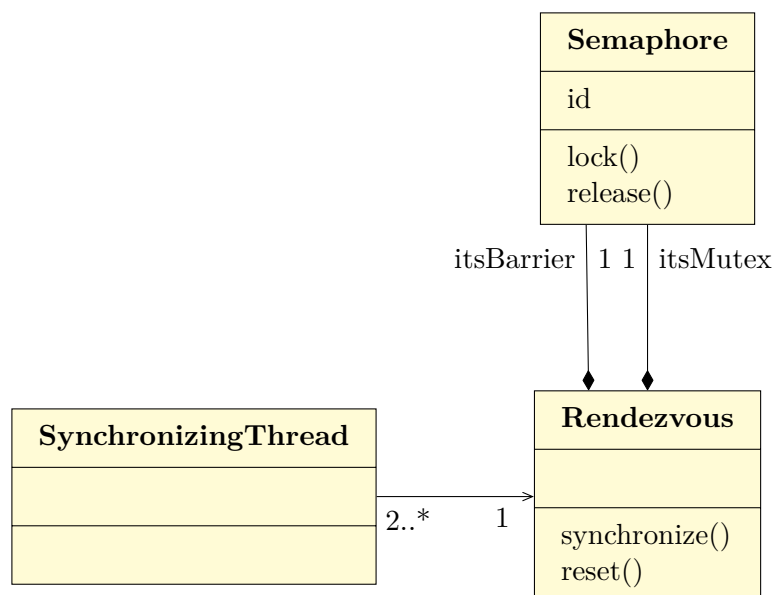
این الگو یک روشی برای استفاد سریال از داده‌ها فراهم می‌کند. وجود Mutex در این الگو باعث می‌شود که خود MessageQueue آلوده نشود. به دلیل استفاده از ارتباطات آسنکرون، دریافت داده‌ها در این الگو سریع‌تر از الگوی Guarded Call است. یکی از نکاتی که باید به آن توجه کرد، انتخاب درست اندازه صف است. اگر صف خیلی 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 سنکرون شدن 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 سنکرون سازی تسکها، تعدادی شرط تعریف می‌کند که با انجام آنها، تسکها سنکرون شده و آزاد می‌شوند. این کار به این صورت انجام می‌شود که هر یک از تسکها خود را پیش شیء **Rendezvous** رجیستر کرده و تا زمانی که این کلاس تصمیم بگیرد متوقف می‌شوند.



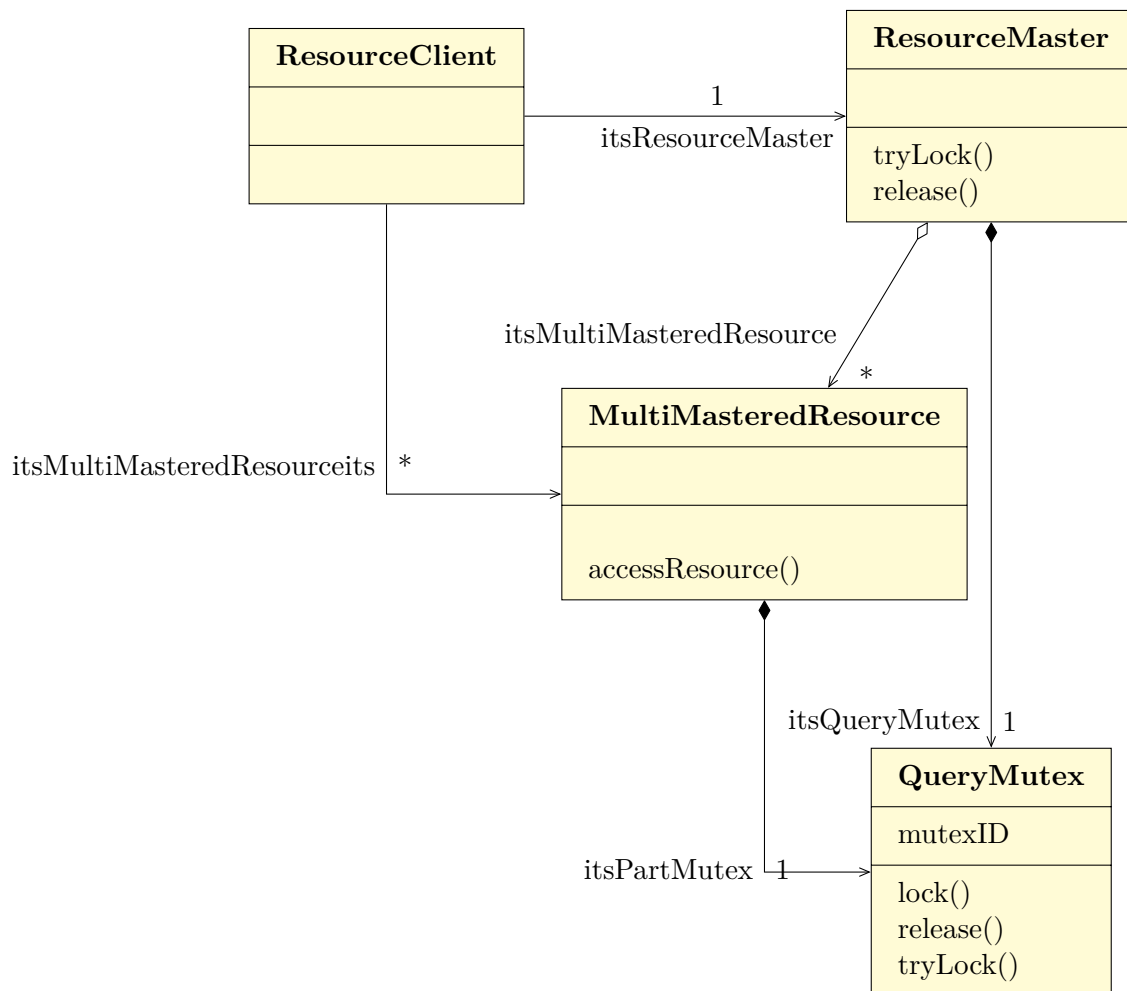
شکل ۱۴: دیاگرام کلاس Rendezvous

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 سنکرون شدن، 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 [Priority Inheritance](#).

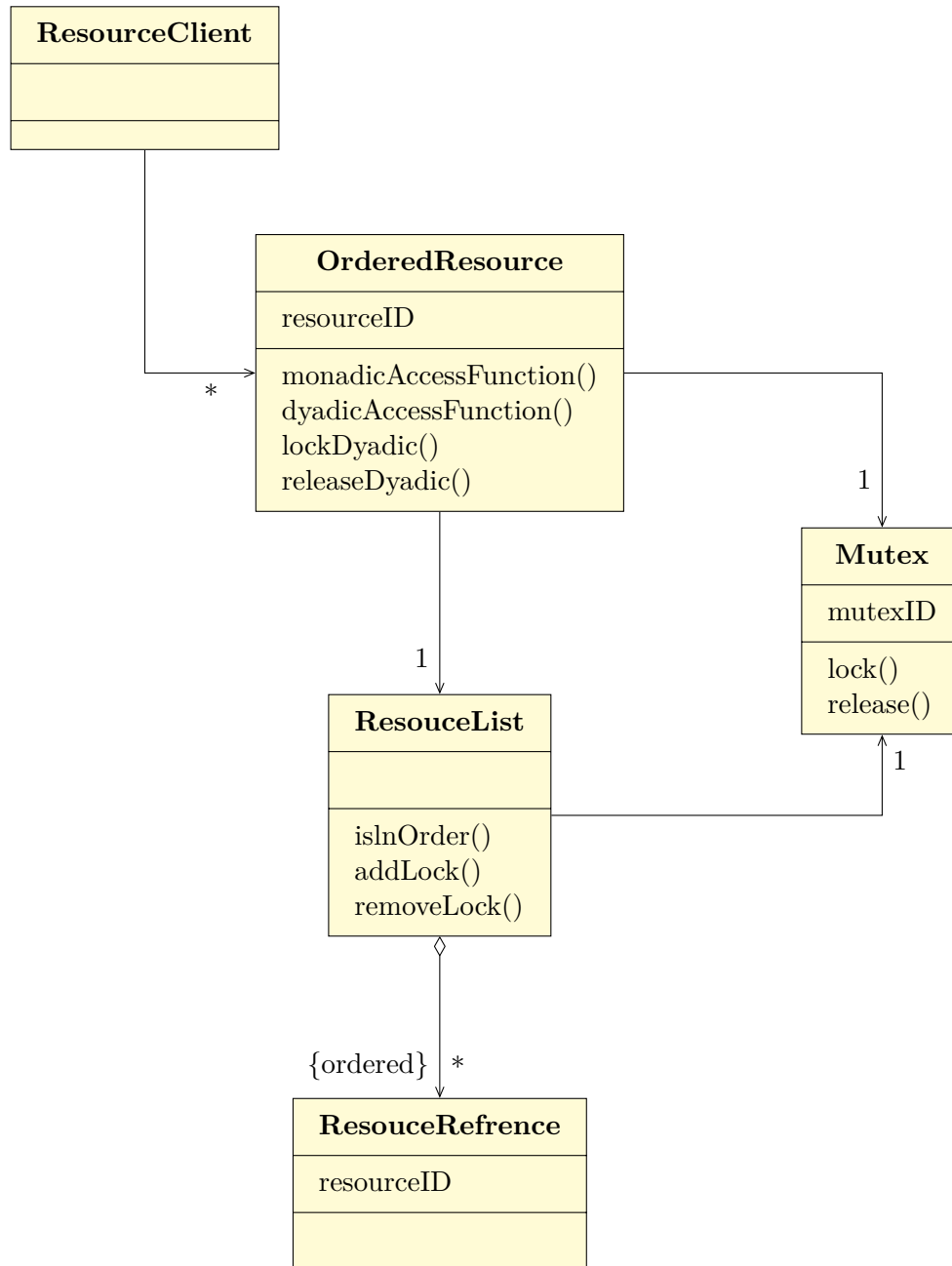


شکل ۱۵: دیگرام کلاس Simultaneous Locking

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 **ResourceManager** را بدست بگیرند، ندارند. زمانی که درخواست قفل کردن برای **ResourceManager** 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 **ResourceManager** 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 دریافت‌کننده علاوه بر نوع رویدادی که رخ داده‌است، باید دارای داده‌های مربوط به رویداد نیز باشد. در این برای تمامی ماشین‌های حالت، تنها یک دریافت‌کننده 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 دریافت‌کننده 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 ماشین‌های حالت با تعداد حالت‌های بسیار زیاد استفاده می‌شود. این الگو با ساخت یک جدول دوبعدی از نحوه گذار حالت‌ها از حالتی به حالت دیگر، ساختار ماشین‌های 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، هر کدام با ۳ حالت داشته باشیم، این الگو این ساختار را به یک ماشین حالت به فرم 3×3 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 اضافه شدن 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 ماشین‌های حالت 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.

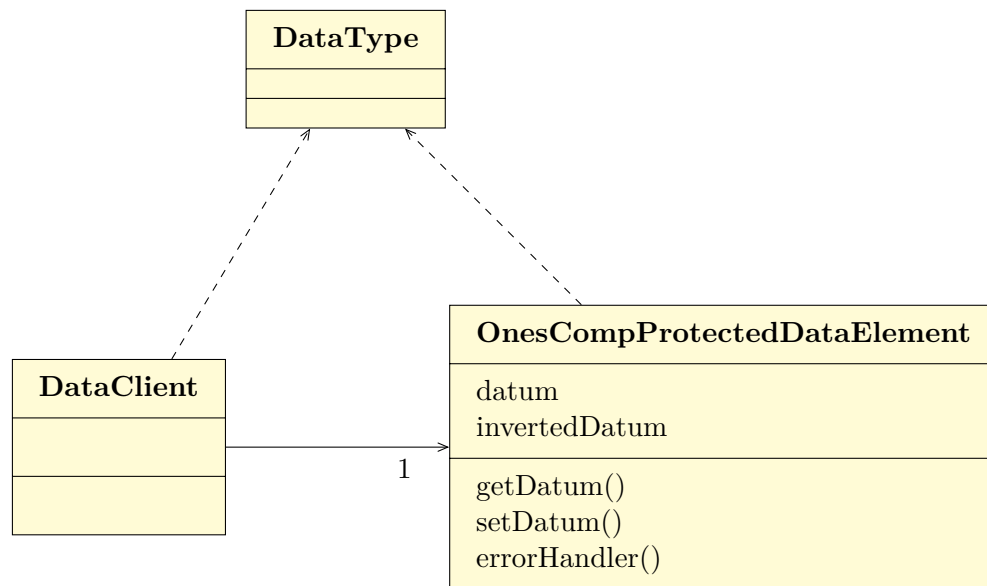
۴.۲ الگوهای امنیت و قابلیت اطمینان

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 گرفته شده. این الگوها به طور کلی با اضافه کردن 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 داده باشد 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 گرفته شده آن دقیقاً قرینه بودند، آن‌گاه داده بدون خطا ذخیره شده است و اگر اینگونه نباشد، نوشتن این داده با خطا مواجه شده بود.

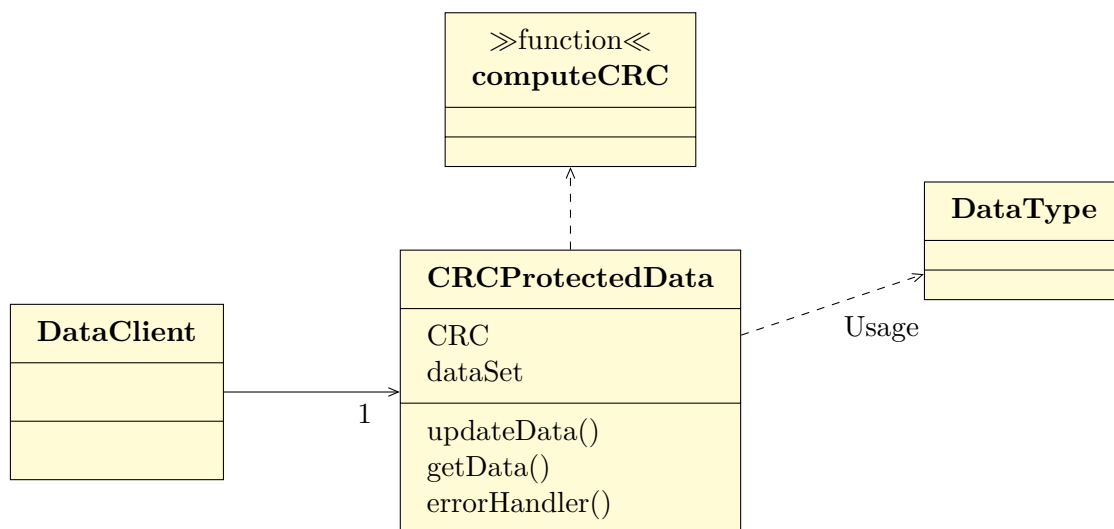


شکل ۱۷: دیاگرام کلاس One's Complement

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 نوشته شده در زمان خوانده شدن نیز باعث سربار اضافی می‌شوند. با این حال این الگو می‌تواند خطاهای گفته شده 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.



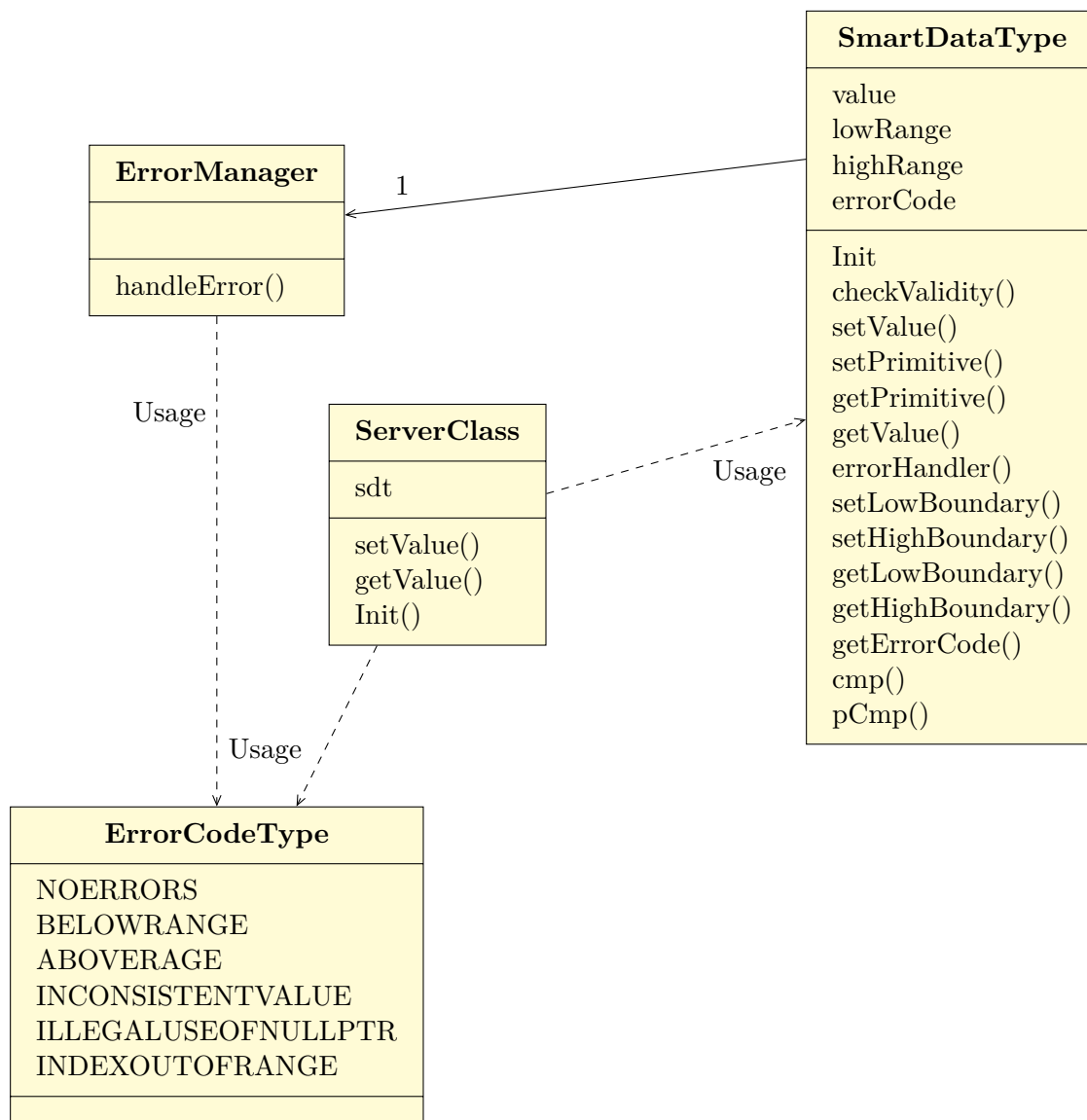
شکل ۱۸: دیاگرام کلاس CRC

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 هندل کردن 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.

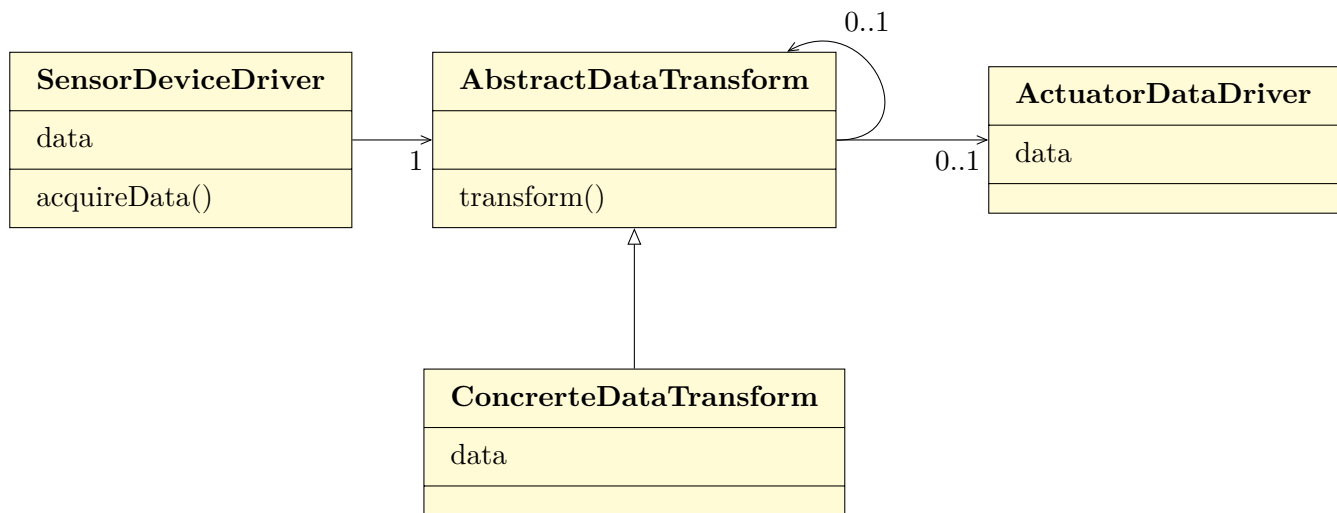


شکل ۱۹: دیاگرام کلاس Smart Data

همانطور که در شکل ۱۹ دیده می‌شود، ساختار داده **SmartDataType**، توابع زیادی برای چک کردن داده خود به **ServerClass** ارائه داده‌است. این الگو با قراردادن چک‌های امنیتی بر روی ساختار داده، می‌تواند باعث امنیت داده در زمان `set` شدن شود. اما این چک‌ها خود سرباری 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

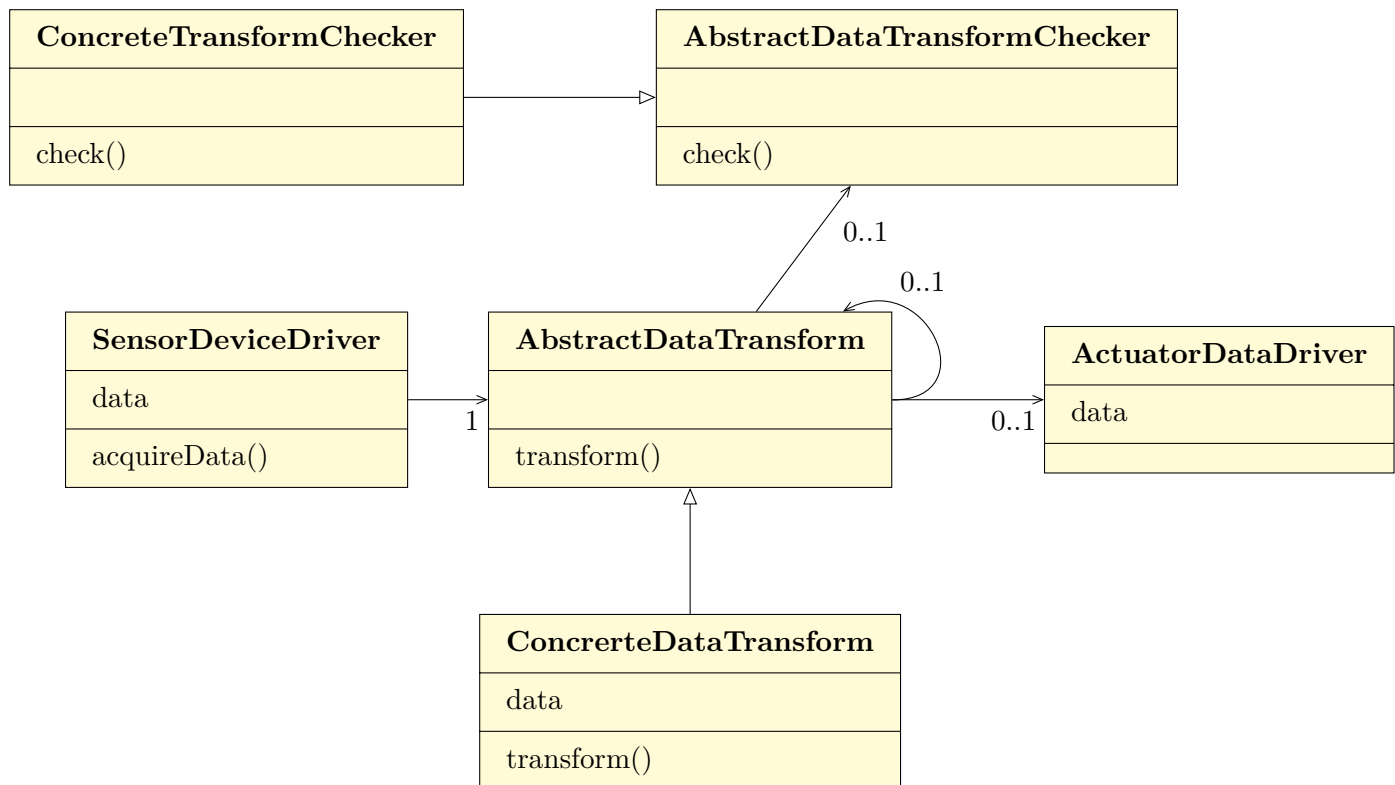
این الگو [۱] بر پایه الگوی Channel ایجاد شده است. این الگو با پیاده‌سازی تعدادی چک و گاردهای تعبیه‌شده در بخش‌های مهم کانال، می‌خواهد مراتبی برای ایجاد امنیت بسازد. میزان تکرار داده در این الگو کمتر از الگوی Channel است و به همین دلیل در صورت تشخیص خطا، نمی‌تواند به کار خود ادامه دهد اما در صورتی که 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 سوییچ کردن روی یک کانال دیگر، سیستم را از حالت خطا خارج می‌کند. همانطور که در شکل ۲۲ مشخص است، این الگو کاملاً مشابه ساختار الگوی Protected Single Channel است با این تفاوت که کلاس ConcreteTransformChecker به دو کانال متصل است. یکی کانال فعلی با 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 شده است.



شکل ۲۱: دیاگرام کلاس Protected Single Channel

۱.۵.۲ الگوی Layered

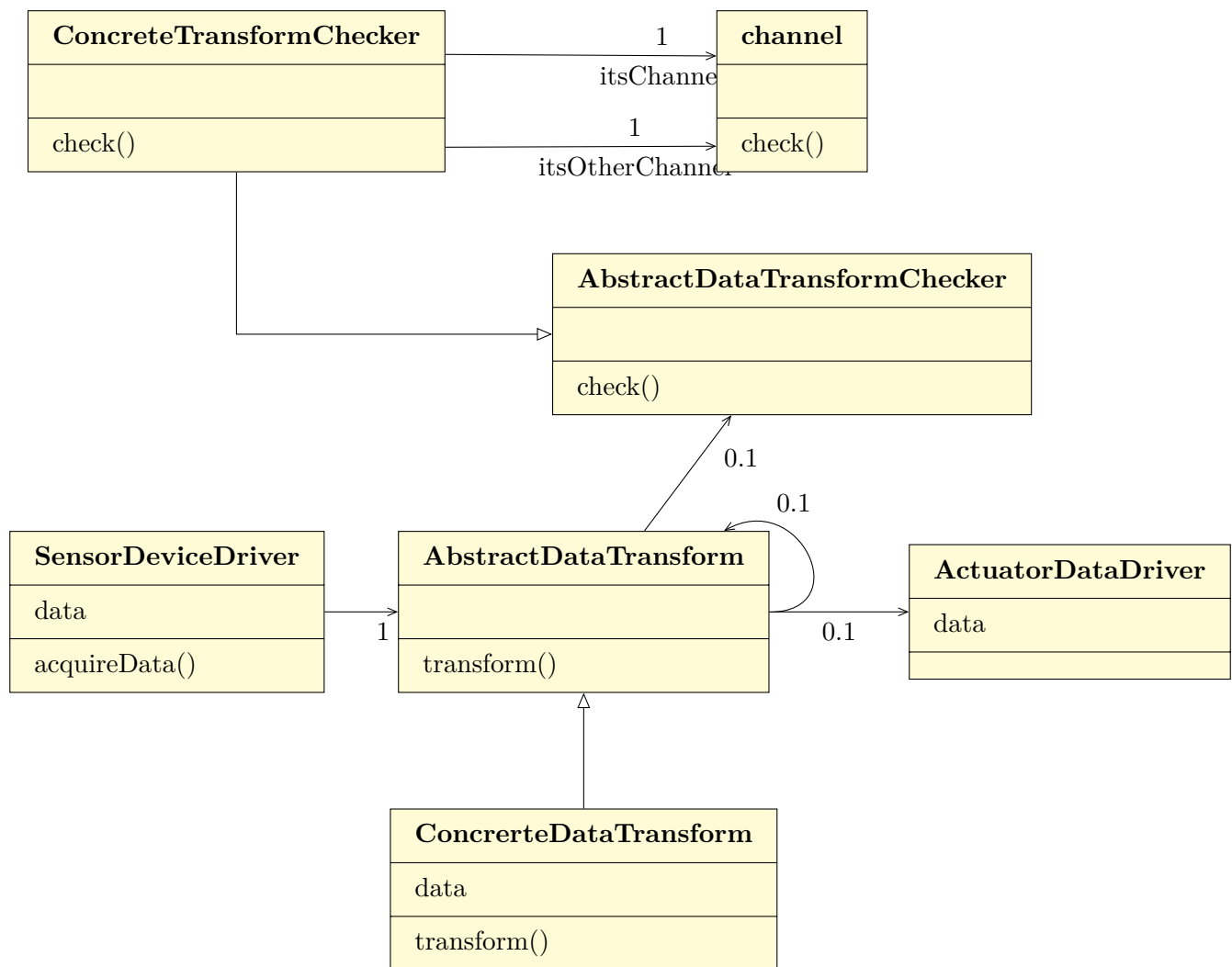
الگوی لایه‌ای [۴] دامنه‌های سیستم را بر اساس سطوح انتزاعی مختلف به صورت سلسله‌مراتبی سازمان‌دهی می‌کند. مفاهیم انتزاعی‌تر در یک دامنه با استفاده از مفاهیم ملموس‌تر در دامنه‌های دیگر پیاده‌سازی می‌شوند. این ساختار به متخصصان اجازه می‌دهد تا به طور موثر در زمینه تخصصی خود کار کنند بدون اینکه نیاز به درک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

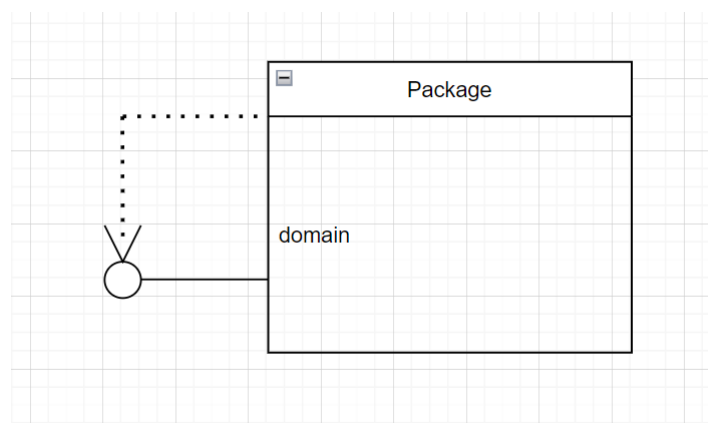
الگوی معماری پنج‌لایه [۴] یک تطبیق خاص از الگوی Layered است که برای ساختاردهی بسیاری از سیستم‌های نهفته و بی‌درنگ مفید است. این الگو معماری منطقی را به پنج لایه تقسیم می‌کند که این امر به توسعه‌دهندگان کمک می‌کند تا به راحتی ساختار سیستم‌های جدید را درک کنند. این الگو از قابلیت انتقال بین پلتفرم‌های 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.



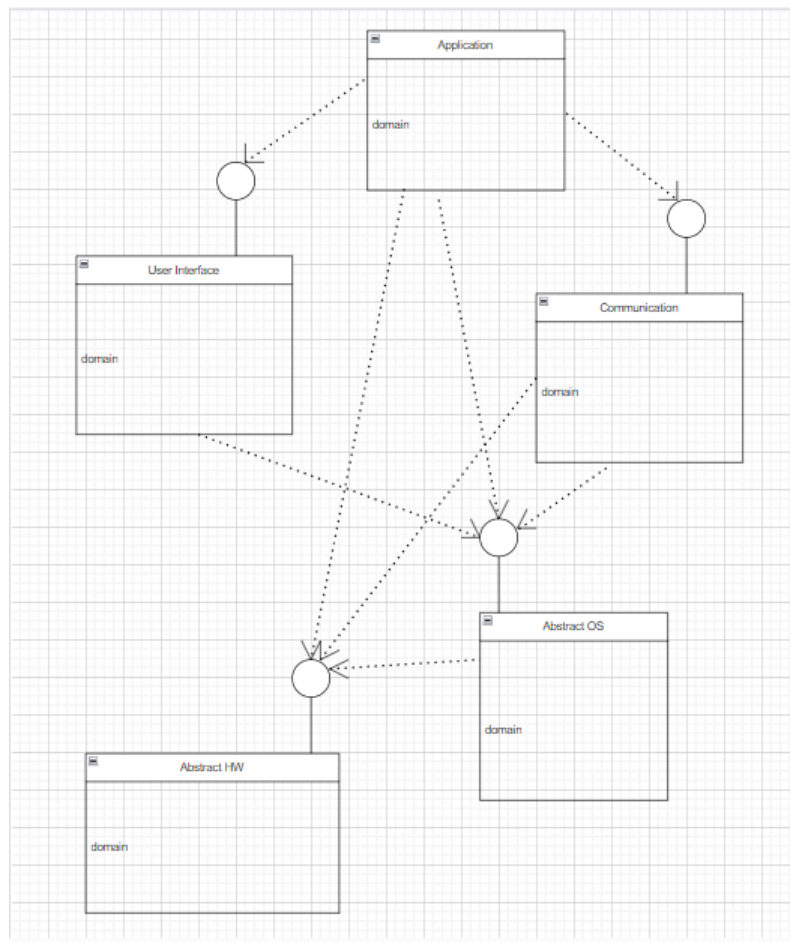
شکل ۲۲: دیاگرام کلاس Dual Channel



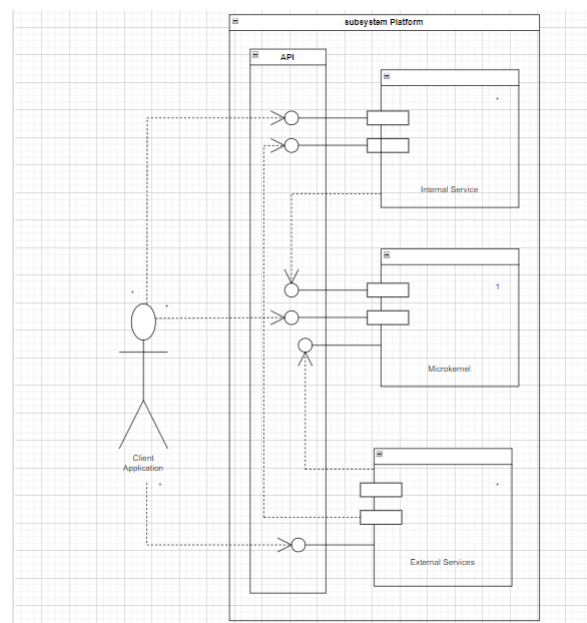
شکل ۲۳: ساختار الگوی Layered

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

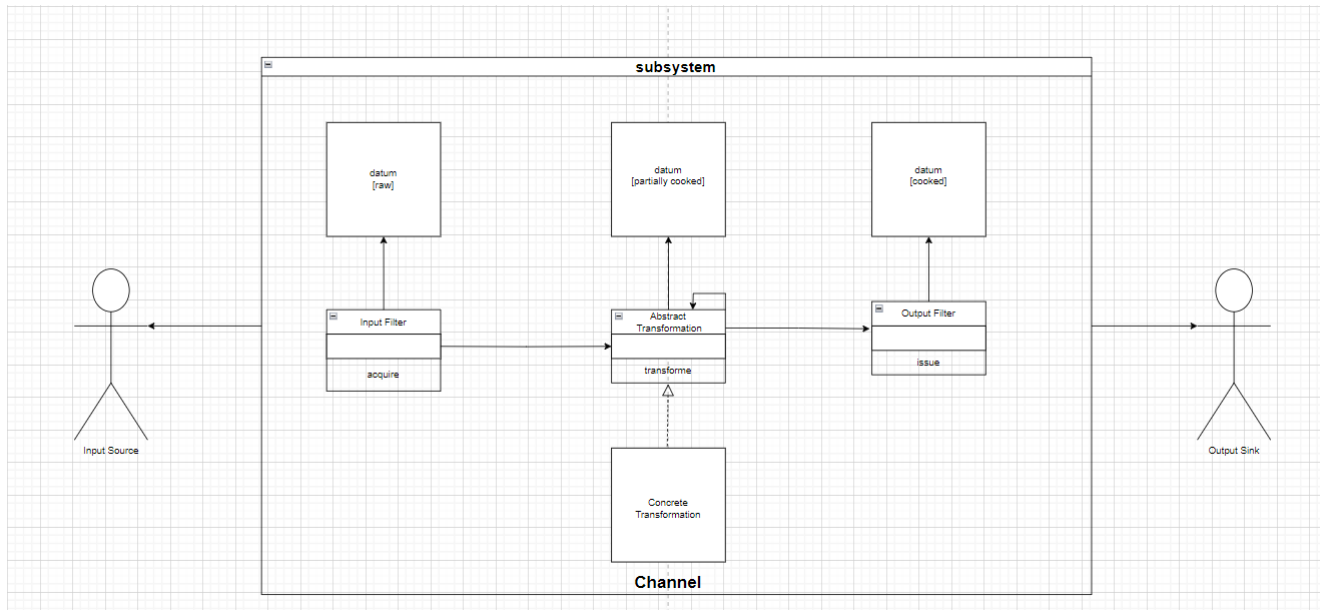


شکل ۲۴: ساختار الگوی Five Layer



شکل ۲۵: ساختار الگوی Microkernel

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.



شکل ۲۶: ساختار الگوی Channel

۵.۵.۲ الگوی Recursive Containment

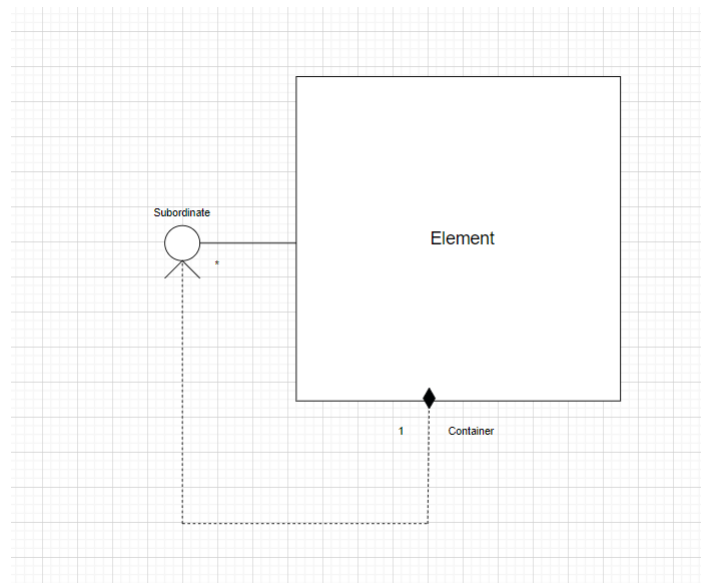
این الگو [۴] برای مدیریت سیستم‌های بسیار پیچیده با نیازمندی‌های فراوان مؤثر است. این الگو شامل شکستن سیستم به اجزای مرتبط در سطوح مختلف جزئیات است، مانند استفاده از میکروسکوپ با سطوح مختلف بزرگ‌نمایی. در هر سطح، اشیاء واسطه‌هایی برای هم‌تایان خود فراهم می‌کنند و وظایف را به اجزای کوچک‌تر داخلی محول می‌کنند، 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

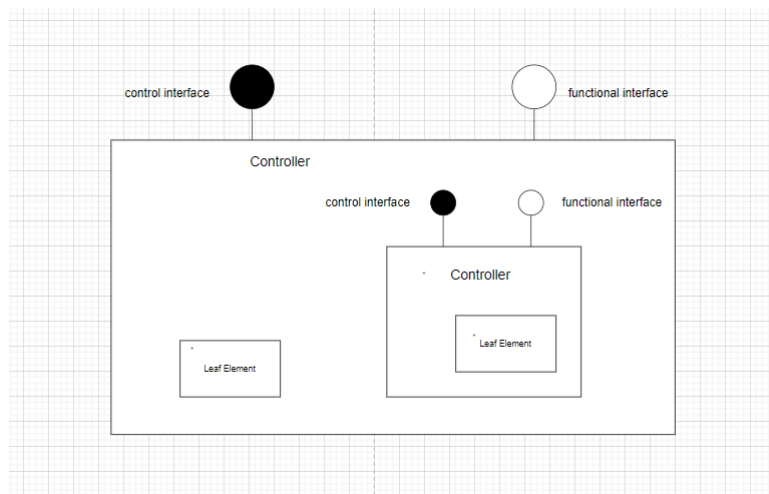
الگوی کنترل سلسله‌مراتبی [۴] یک نسخه تخصصی از الگوی Recursive Containment است که الگوریتم‌های پیچیده کنترلی را بین اجزای مختلف توزیع می‌کند. این الگو از دو نوع واسطه استفاده می‌کند: واسطه‌های کنترلی که نحوه دستیابی به رفتارها را نظارت و کنترل می‌کنند و واسطه‌های عملکردی که خدمات کنترل‌شده توسط واسطه‌های دیگر را فراهم می‌کنند. واسطه‌های کنترلی کیفیت خدمات، مانند دقت و صحت، را تعیین می‌کنند و سیاست‌های اجرایی را تنظیم می‌کنند. واسطه‌های عملکردی رفتار مطلوب را با استفاده از کیفیت خدمات و سیاست‌های تنظیم شده توسط واسطه 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



شکل ۲۷: ساختار الگوی Recursive Containment

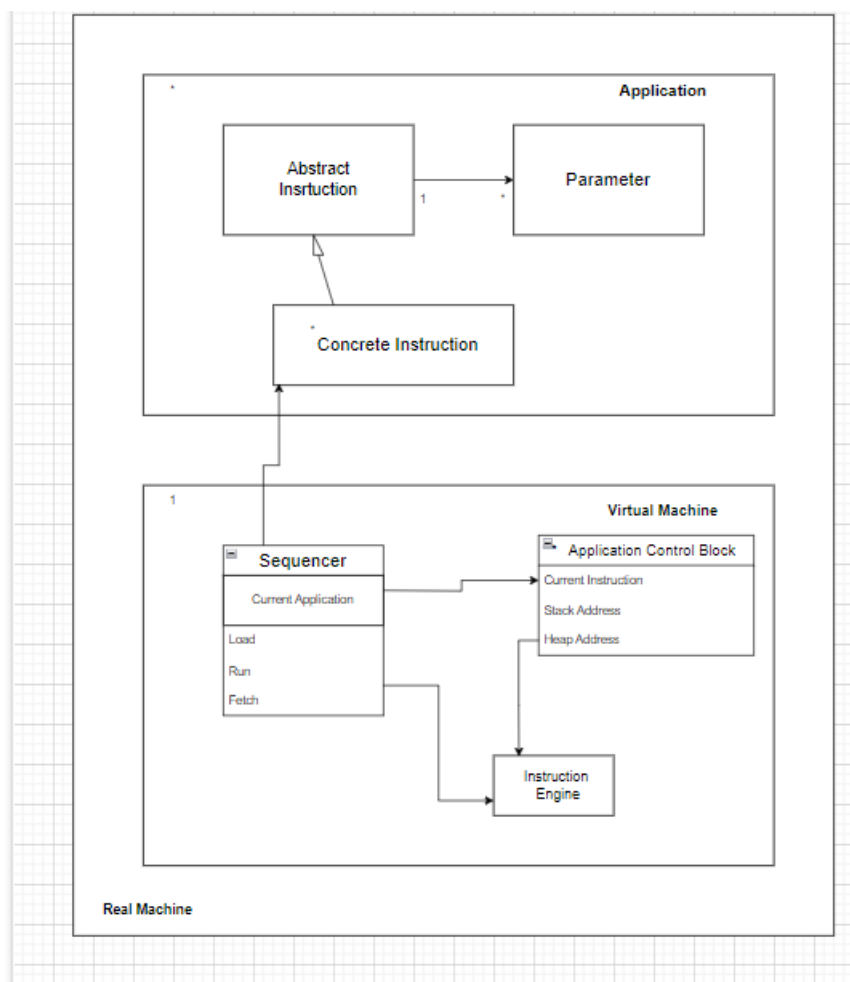


شکل ۲۸: ساختار الگوی Hierarchical Control

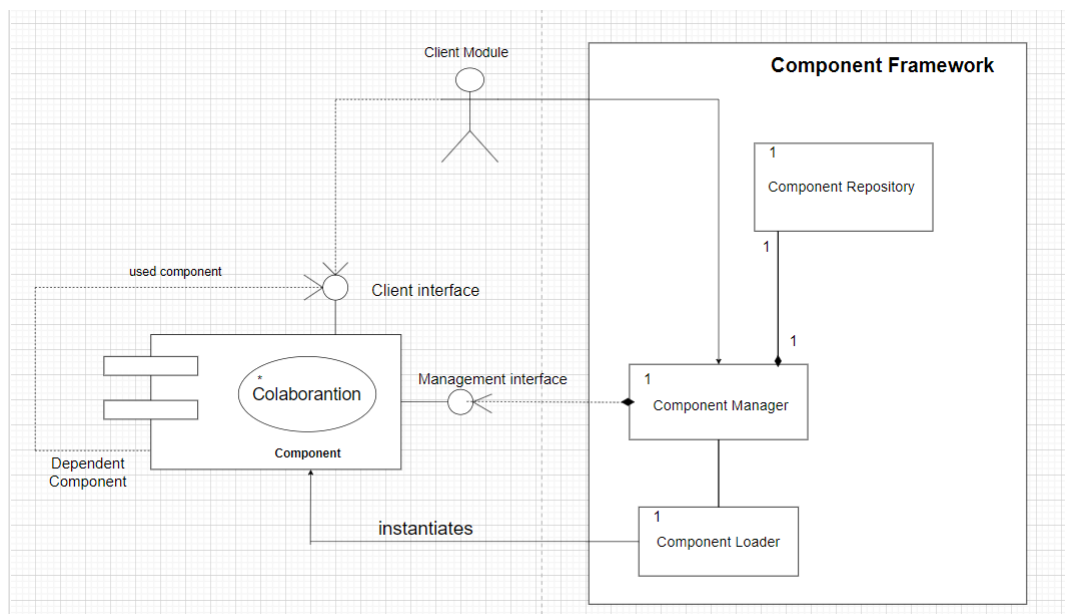
شوند، اما مزایای آن شامل ساده سازی انتقال و اندازه کوچکتر برنامه ها به دلیل اشتراک کتابخانه ها در داخل ماشین مجازی است. با این حال، ماشین های مجازی می توانند منابع زیادی مصرف کنند و ممکن است برای دستگاه های با محدودیت حافظه مناسب نباشند. در چنین شرایطی ممکن است الگویی مانند الگوی [Microkernel](#) مناسب تر باشد.

۸.۵.۲ معماری Component-Based

در UML، یک Component یک اثر زمان اجرا و یک واحد قابل جایگزینی اساسی در نرم افزار است که مشابه یک شیء بزرگ مقیاس شامل اشیاء کوچکتری است که واسط آن را پیاده سازی می کنند. Component ها دارای کپسوله سازی قوی و واسط های مستقل از زبان برنامه نویسی و کاملاً تعریف شده هستند. سیستم های مبتنی بر Component که از این اشیاء بزرگ مقیاس به عنوان واحدهای معماری استفاده می کنند، از نگهداری آسان، جداسازی عیوب، استقلال از زبان منبع، سادگی توسعه و قابلیت استفاده مجدد بهره مند می شوند. Component ها معمولاً اشیاء کوچکتری را برای هدف رفتاری مشترک در زمان اجرا جمع می کنند. آنها دارای واسط های مبهم هستند، به این معنا که جزئیات داخلی آنها از کلاسیت مخفی است که این امر جایگزینی را تضمین می کند اما ممکن است منجر به ناکارآمدی شود. الگوی معماری مبتنی بر Component [۴] معماری سیستم را قوی و قابل استفاده مجدد می سازد اما ممکن است به دلیل استفاده از کل Component ها حتی اگر فقط بخشی از عملکرد آنها استفاده شود، منابع اضافی مصرف کند.



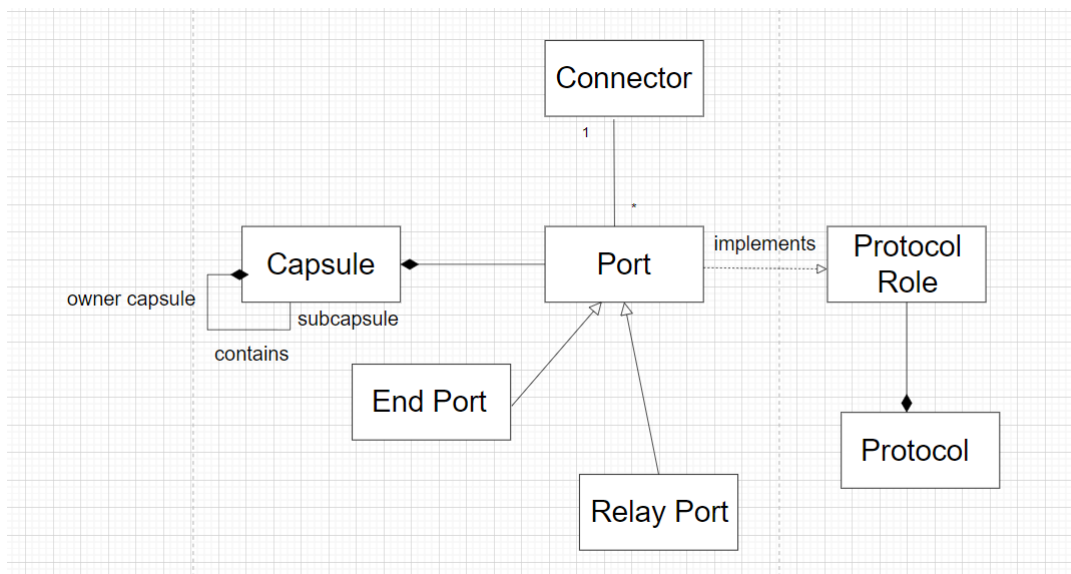
شکل ۲۹: ساختار الگوی Virtual Machine



شکل ۳۰: ساختار الگوی Component-Based

۹.۵.۲ الگوی ROOM

ROOM^۴ [۴] یک روش قدیمی تر است که پیش از UML وجود داشته است، با این حال UML می تواند ROOM را مدل سازی کند، همان طور که توسط تطابق UML-RT نشان داده شده است. ROOM نقش های خاصی برای رابط های دوطرفه به نام پورت ها شناسایی می کند و از کلاس های پروتکل برای کنترل این تعاملات استفاده می کند و کیسوله سازی قوی ارائه می دهد. این روش از نمودارهای حالت برای اجرای رفتار استفاده می کند و برای سیستم هایی با تعاملات پیچیده بین اشیاء درشت دانه مناسب است، چه توزیع شده باشند یا نه. این متدولوژی کیسول ها را معرفی می کند که می توانند زیرکیسول ها را شامل شوند و از پورت های رله برای ارسال پیام استفاده کنند. با وجود مزایای آن در مدیریت رابط ها و تعاملات پیچیده، ماهیت سنگین ROOM می تواند روابط ساده را پیچیده کند و باید با دقت اعمال شود تا از محدودیت بیش از حد جلوگیری شود.



شکل ۳۱: ساختار الگوی ROOM

۶.۲ الگوهای معماری همزمانی

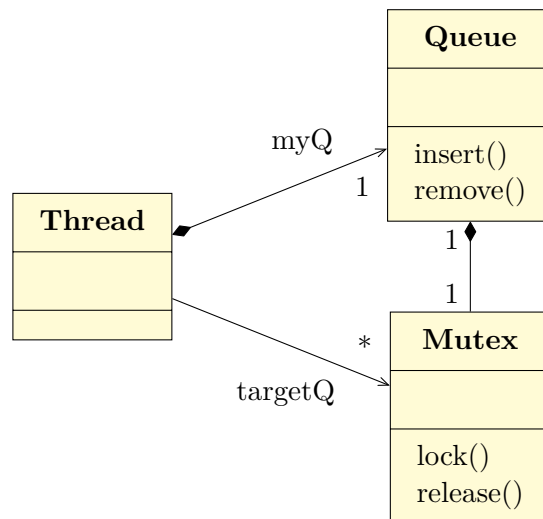
همانطور که Douglass در [۴] می گوید، یکی از مسائل مهم در معماری سیستم ها، کنترل و زمان بندی بخش های مختلف معماری سیستم است. در UML، همزمانی از طریق Thread ها مدیریت می شود که هر Thread در یک شیء «فعال» ریشه دارد و اشیاء «غیرفعال» را مدیریت می کند. توسعه دهندگان Thread ها را شناسایی کرده و هر کدام را به یک شیء «فعال» اختصاص می دهند. این اشیاء فعال مدیریت پیام و اجرا را در Thread های خود انجام می دهند. این بخش نیز براساس [۴] نوشته شده است.

۱.۶.۲ الگوی Message Queuing

الگوی صف پیام [۴] روشی ساده برای ارتباط بین Thread ها فراهم می کند. علی رغم اینکه این روش نسبتاً سنگین برای اشتراک گذاری اطلاعات است، اما به طور گسترده استفاده می شود زیرا توسط بیشتر سیستم عامل ها پشتیبانی می شود و به راحتی قابل اثبات صحت است. این الگو از مشکلات Mutual Exclusion جلوگیری می کند زیرا هیچ منبع اشتراکی نیاز به محافظت ندارد، که همگام سازی را ساده کرده و یکپارچگی داده ها را تضمین می کند. در این الگو، اطلاعات به جای ارجاع، به صورت مقدار پاس داده می شوند و از مسائل فساد داده ای که در سیستم های همزمان رایج است، جلوگیری می شود. با این حال، این روش برای پردازش ساختارهای داده بزرگ کارایی کمتری دارد و اشتراک گذاری اطلاعات بسیار کارآمد را تسهیل نمی کند. (این الگو همان الگوی **Queueing** است که در [۱] گفته شده.)

۲.۶.۲ الگوی Interrupt

وقفه ها [۴] به دلیل کارایی و اجرای سریع شان بسیار توصیه می شوند و در سیستم های بی درنگ و نهفته برای پاسخ به رویدادهای اضطراری ضروری هستند. این وقفه ها در مواقعی که پاسخ ها کوتاه و غیرقابل قطع هستند، عملکرد عالی دارند. با این حال، آنها برای همه موقعیت ها

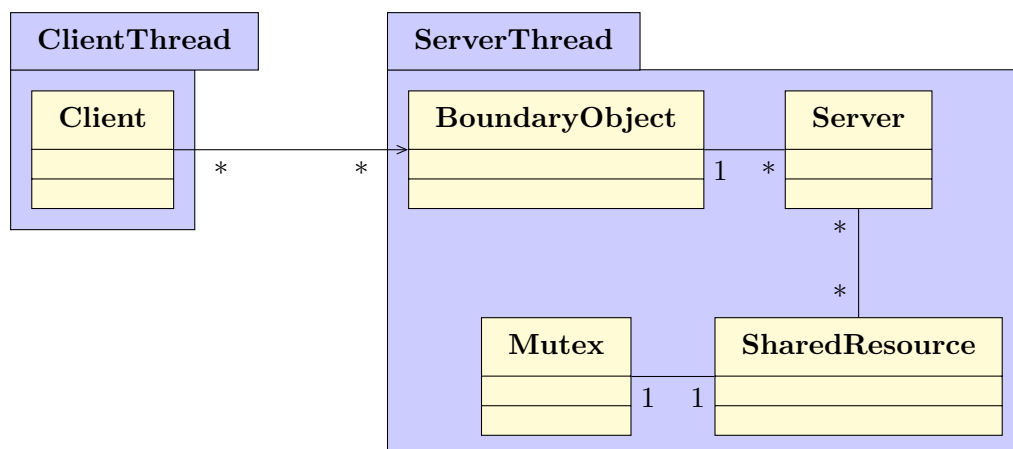


شکل ۳۲: دیاگرام کلاس Message Queuing

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] گفته شده.)

۳.۶.۲ الگوی Guarded Call

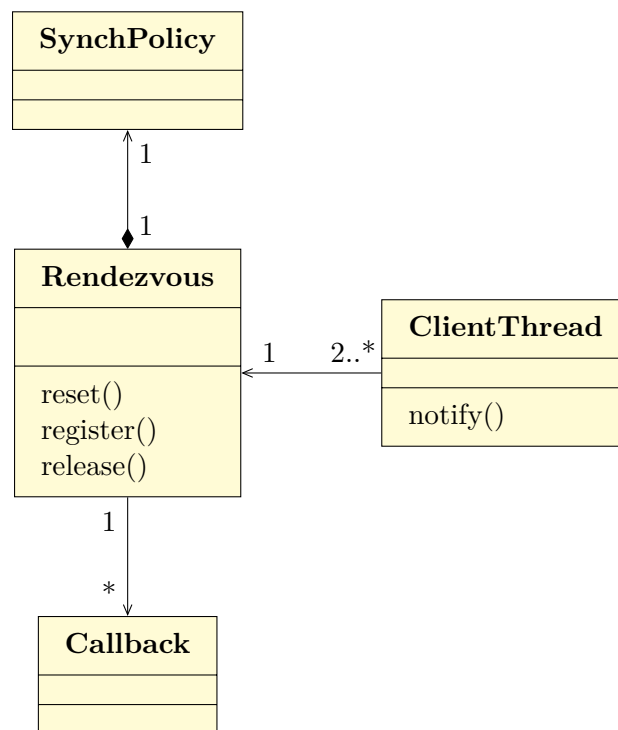
الگوی فراخوانی محافظت‌شده [۴] راهی برای دستیابی به همگام‌سازی به موقع بین Threadها از طریق فراخوانی همزمان متدها در یک Thread دیگر ارائه می‌دهد و از Mutual Exclusion Semaphores برای جلوگیری از فساد داده و Deadlock استفاده می‌کند. در حالی که ارتباطات غیرهمزمان مانند الگوی Message Queuing اغلب منجر به 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 Mutual 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] گفته شده.)



شکل ۳۳: دیاگرام کلاس Guarded Call

۴.۶.۲ الگوی Rendezvous

الگوی Rendezvous [۴] نسخه ساده‌تری از الگوی Guarded Call است که برای همگام‌سازی Thread ها 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 Thread ها 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 Thread ها 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 Thread ها و شروط، مقیاس‌پذیر است. (این الگو همان الگوی Rendezvous است که در [۱] گفته شده).



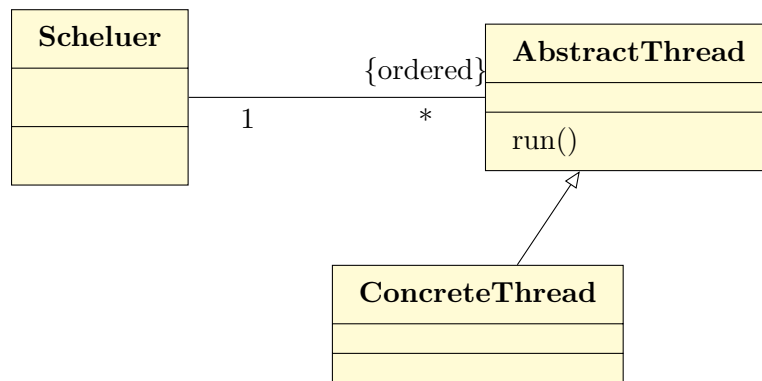
شکل ۳۴: دیاگرام کلاس Rendezvous

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] گفته شده).

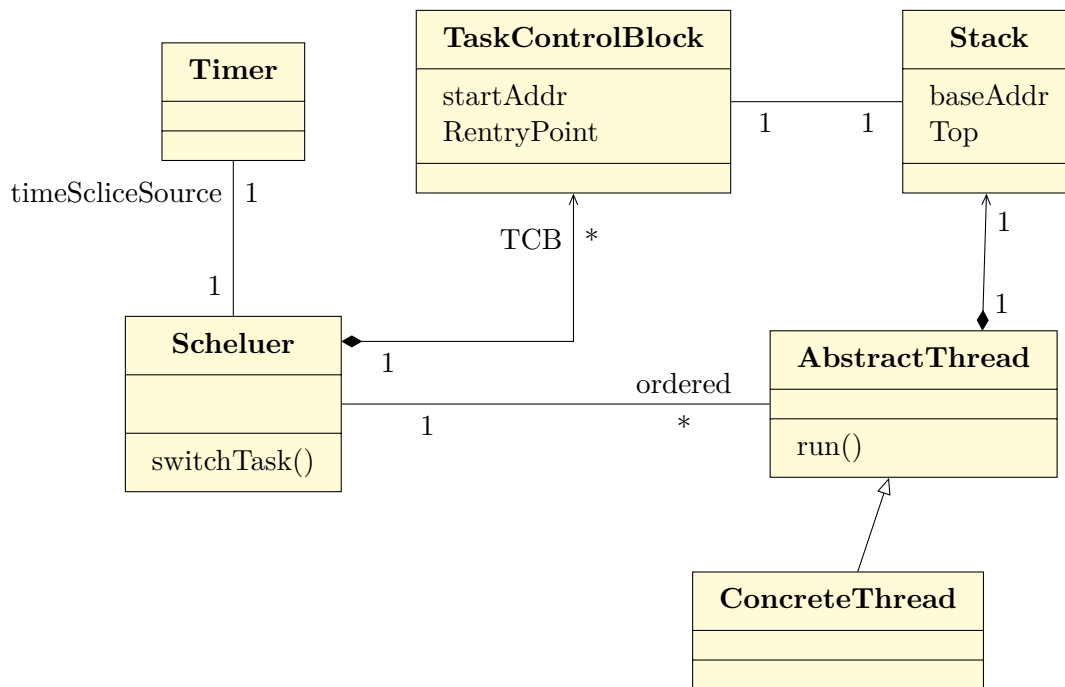
۶.۶.۲ الگوی Round Robin

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



شکل ۳۵: دیاگرام کلاس Cyclic Execution

تشدید وظایف شود و زمان مؤثر هر وظیفه کاهش یابد. مکانیزم‌های اشتراک‌گذاری داده‌ها ابتدایی هستند و پیاده‌سازی مدل‌های پیچیده را دشوار می‌کنند.



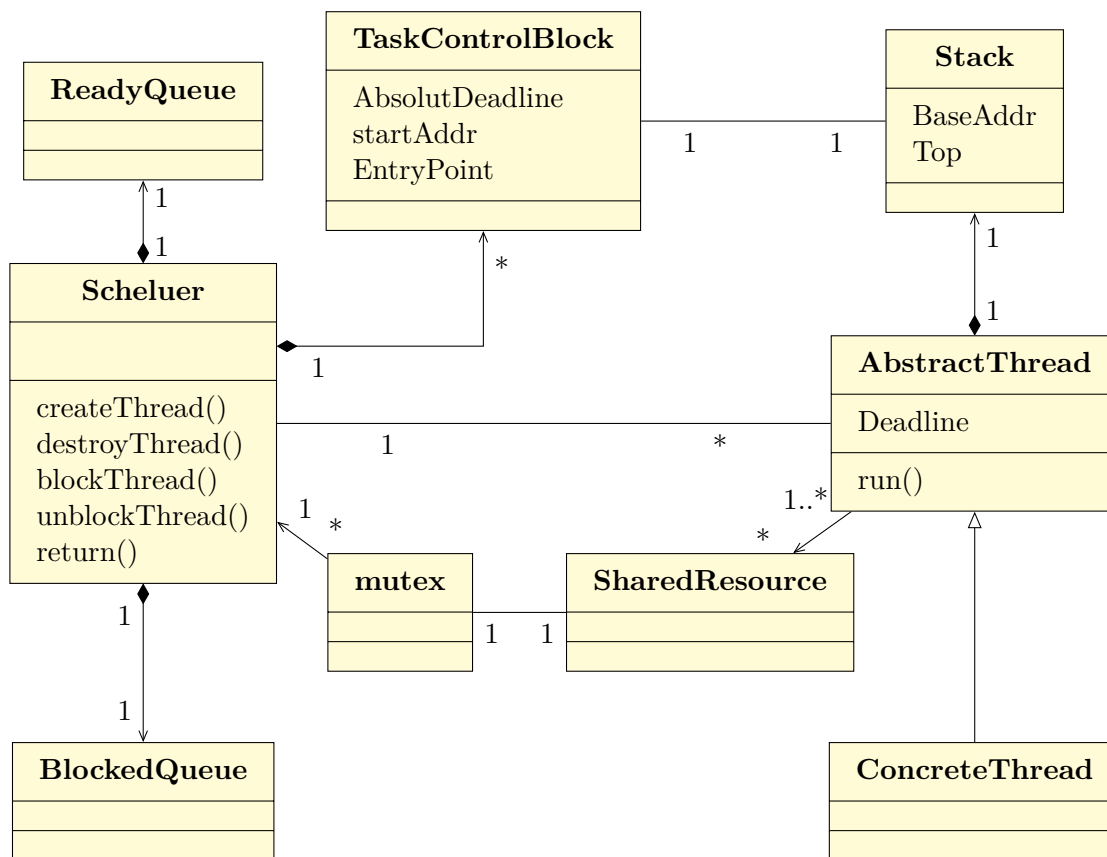
شکل ۳۶: دیاگرام کلاس Round Robin

۷.۶.۲ الگوی Static Priority

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

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.



شکل ۳۷: دیاگرام کلاس Dynamic Priority

۷.۲ الگوهای معماری حافظه

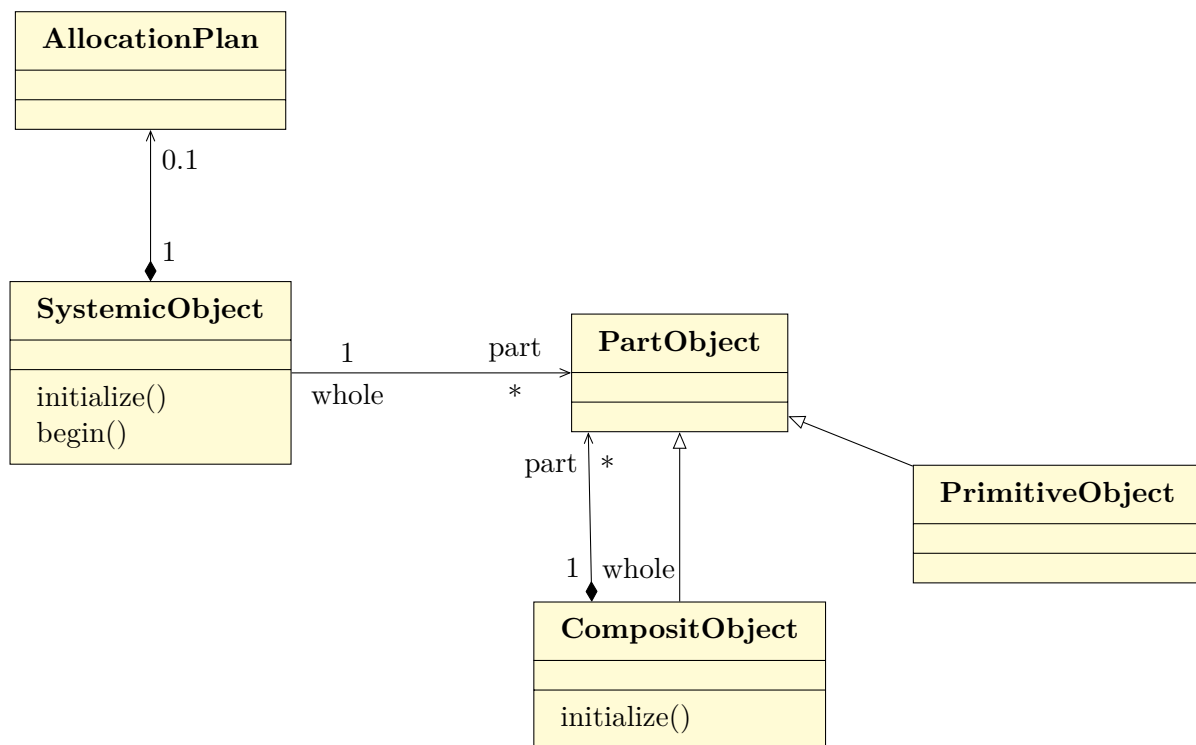
در این بخش به بررسی الگوهای مدیریت حافظه و به اشتراک گذاری منابع در سیستم نرم‌افزاری می‌پردازیم. این دسته از الگوها از [۴] آورده شده‌است.

۱.۷.۲ الگوی Static Allocation

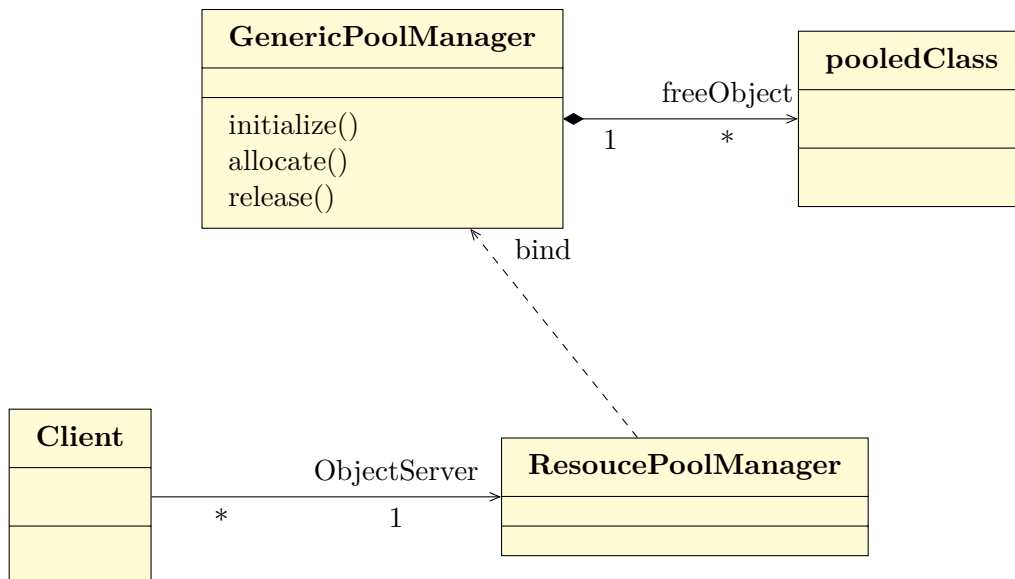
الگوی Static Allocation [۴] برای سیستم‌های ساده با بارهای حافظه قابل پیش‌بینی و ثابت طراحی شده است. این الگو از تخصیص حافظه پویا اجتناب می‌کند تا مشکلاتی مانند زمان‌بندی غیرقابل پیش‌بینی و تکه‌تکه شدن حافظه را از بین ببرد. در عوض، همه اشیاء در زمان راه‌اندازی سیستم تخصیص داده می‌شوند که منجر به زمان 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.



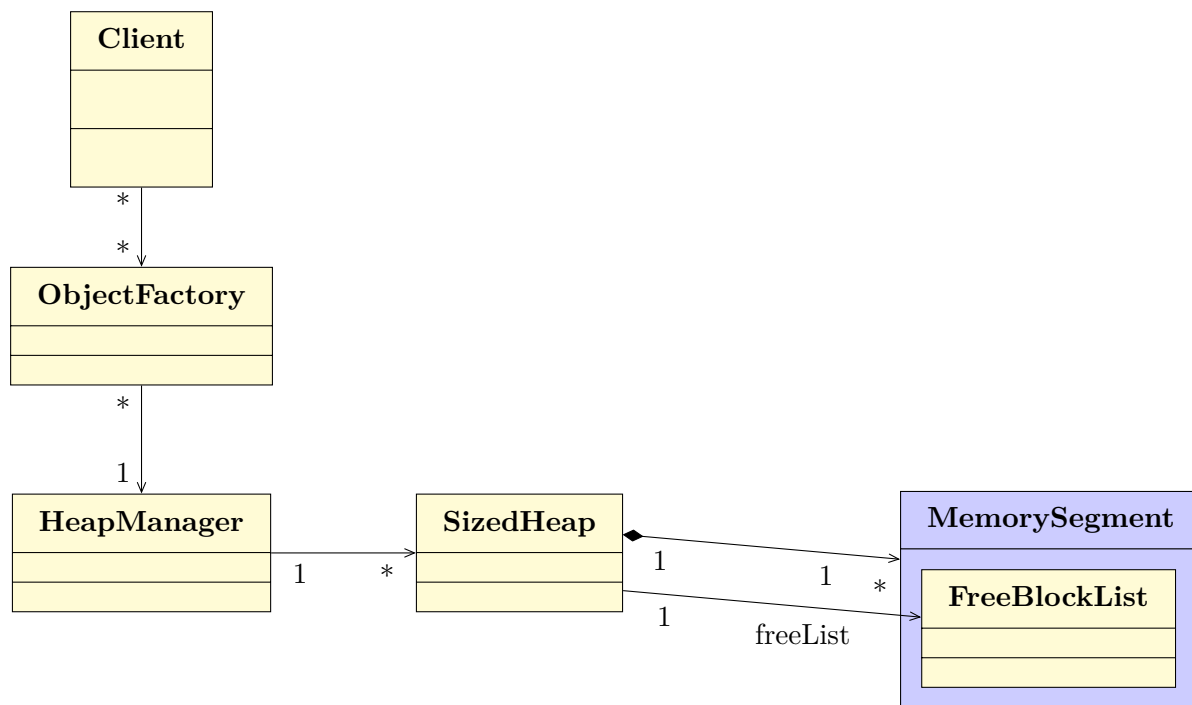
شکل ۳۸: دیاگرام کلاس Static Allocation



شکل ۳۹: دیاگرام کلاس Pool Allocation

۳.۷.۲ الگوی Fixed Sized Buffer

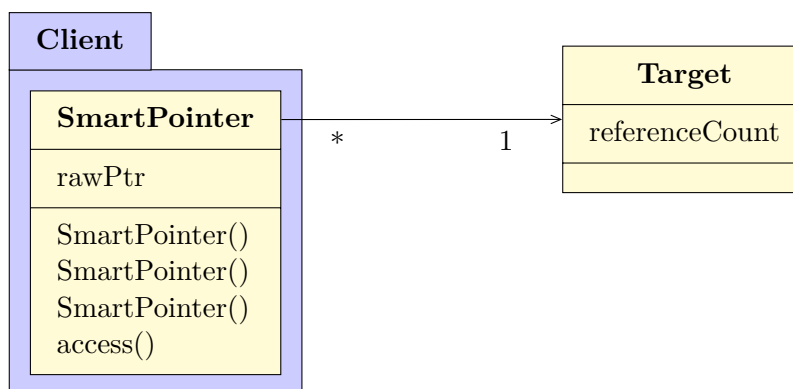
این الگو [۴] مشکل تکه‌تکه شدن حافظه را در تخصیص حافظه پویا برطرف می‌کند، که این مسئله برای سیستم‌های بی‌درنگ نهفته که باید برای مدت‌های طولانی به‌طور قابل اعتماد عمل کنند، بسیار مهم است. این الگو با استفاده از بلوک‌های حافظه با اندازه ثابت از تکه‌تکه شدن جلوگیری می‌کند، اگرچه منجر به هدر رفتن بخشی از 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.



شکل ۴۰: دیگرام کلاس Fixed Sized Buffer

۴.۷.۲ الگوی Smart Pointer

الگوی اشاره گر هوشمند [۴] یک راه حل طراحی برای کاهش مشکلات رایج مربوط به اشاره گرها در برنامه نویسی است، مانند نشت حافظه، Dangling Pointers، اشاره گرهای مقداره‌ی نشده، و نقص‌های محاسباتی اشاره گر. با کپسوله کردن اشاره گرها در اشیاء، اشاره گرهای 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 اشاره گرهای 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.

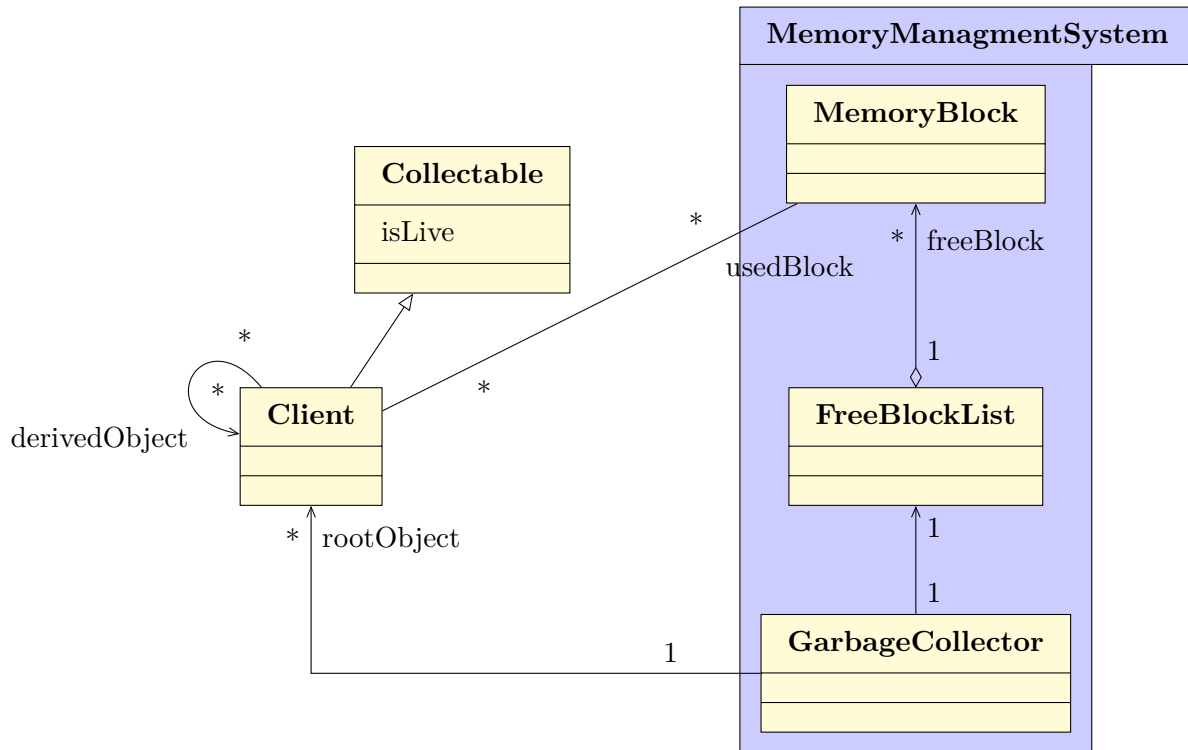


شکل ۴۱: دیگرام کلاس Smart Pointer

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 قابل توجهی مشکلات مرتبط با حافظه را کاهش داده و برای سیستم‌های با دسترسی بالا که نیاز به اجرای طولانی مدت بدون راه‌اندازی مجدد دارند، ایده‌آل است. با این

حال، این الگو به دلیل طبیعت دوره‌ای خود، سر بار 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.



شکل ۴۲: دیاگرام کلاس Garbage Collection

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 اشیای 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 finalize کردن اشیاء دارد.

۸.۲ الگوهای معماری منابع

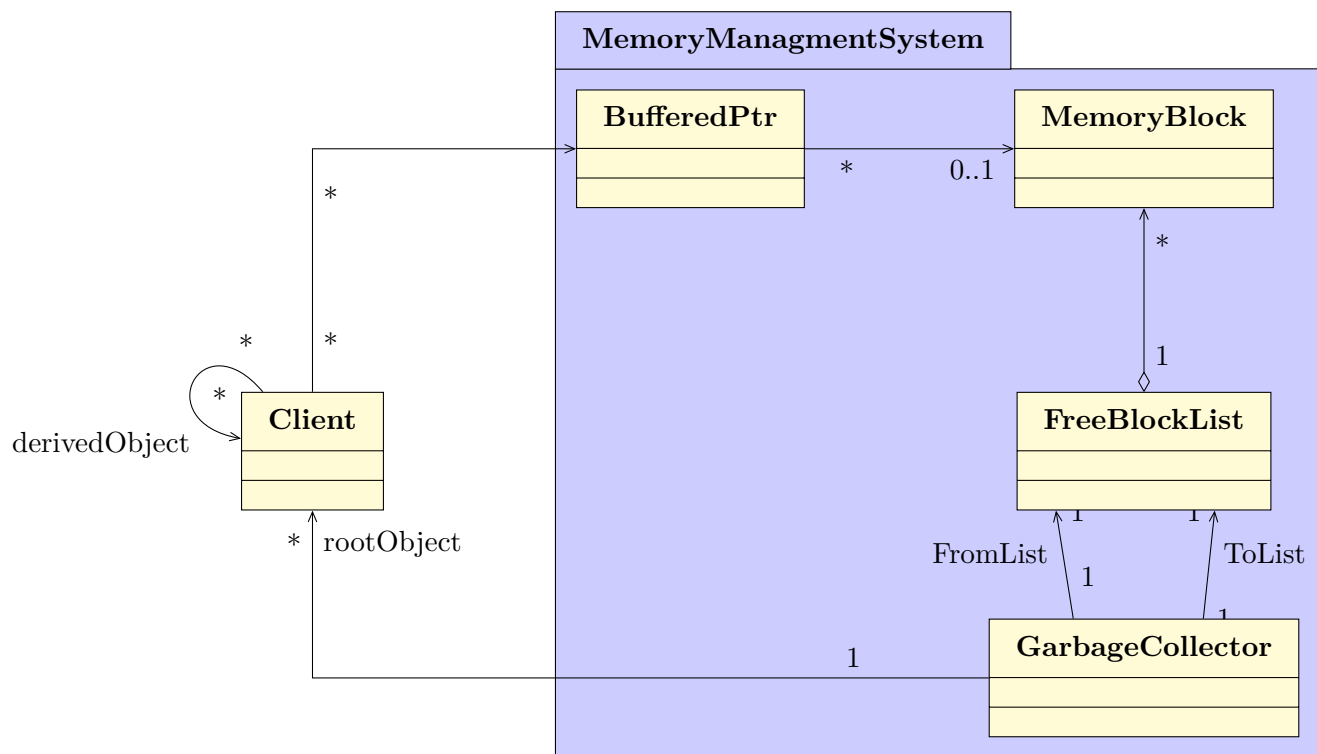
این دسته از الگوها که در [۴] نام برده شده‌اند، به ما می‌گویند که چگونه می‌توان منابع را در یک سیستم بی‌درنگ نهفته به اشتراک گذاشت و مدیریت کرد. در سیستم‌های نهفته، منابع بسیار محدود هستند و این الگوها از این منظر بسیار حائز اهمیت هستند.

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 سر بار 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.



شکل ۴۳: دیاگرام کلاس Garbage Compactor

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.

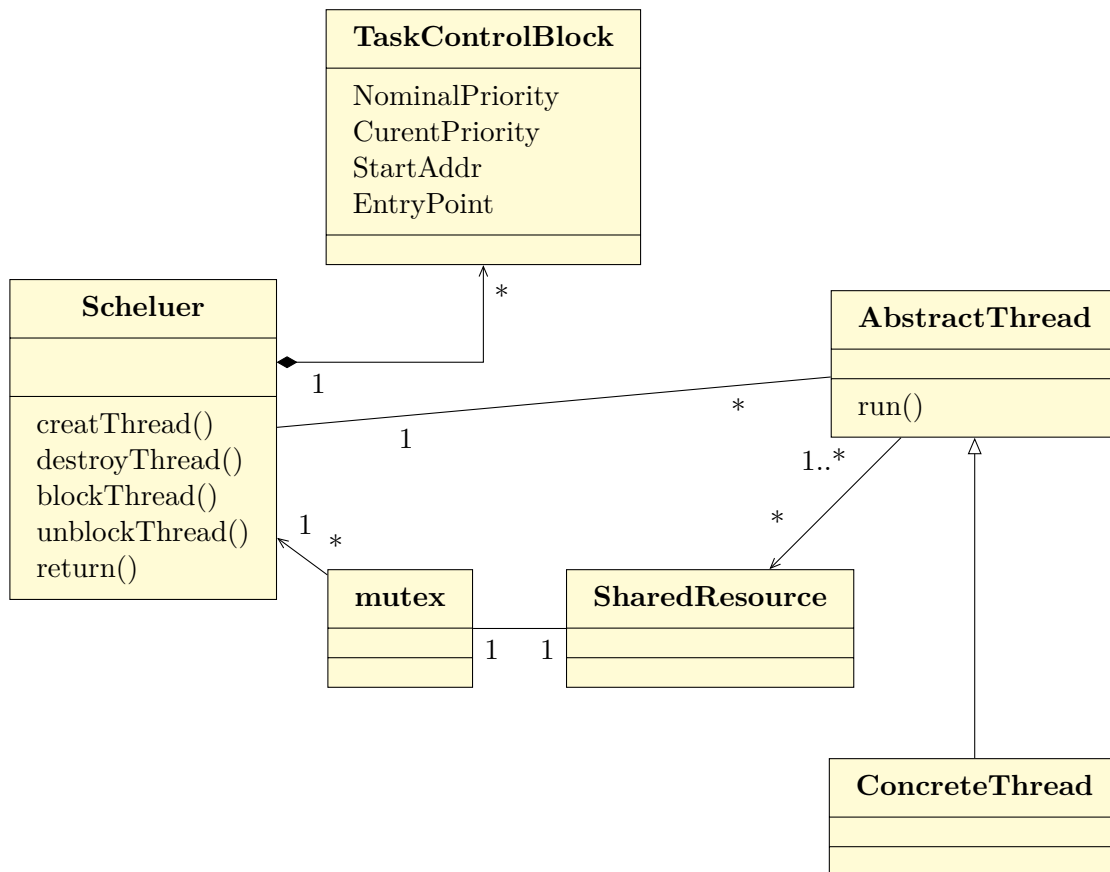
این الگو مشکلات Deadlock را حل نمی‌کند و می‌تواند باعث ایجاد سربار ناشی از جابه‌جایی وظایف و تنظیمات اولویت شود، به ویژه اگر رقابت بر سر منابع بالا باشد.

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.



شکل ۴۴: دیاگرام کلاس Priority Inheritance

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

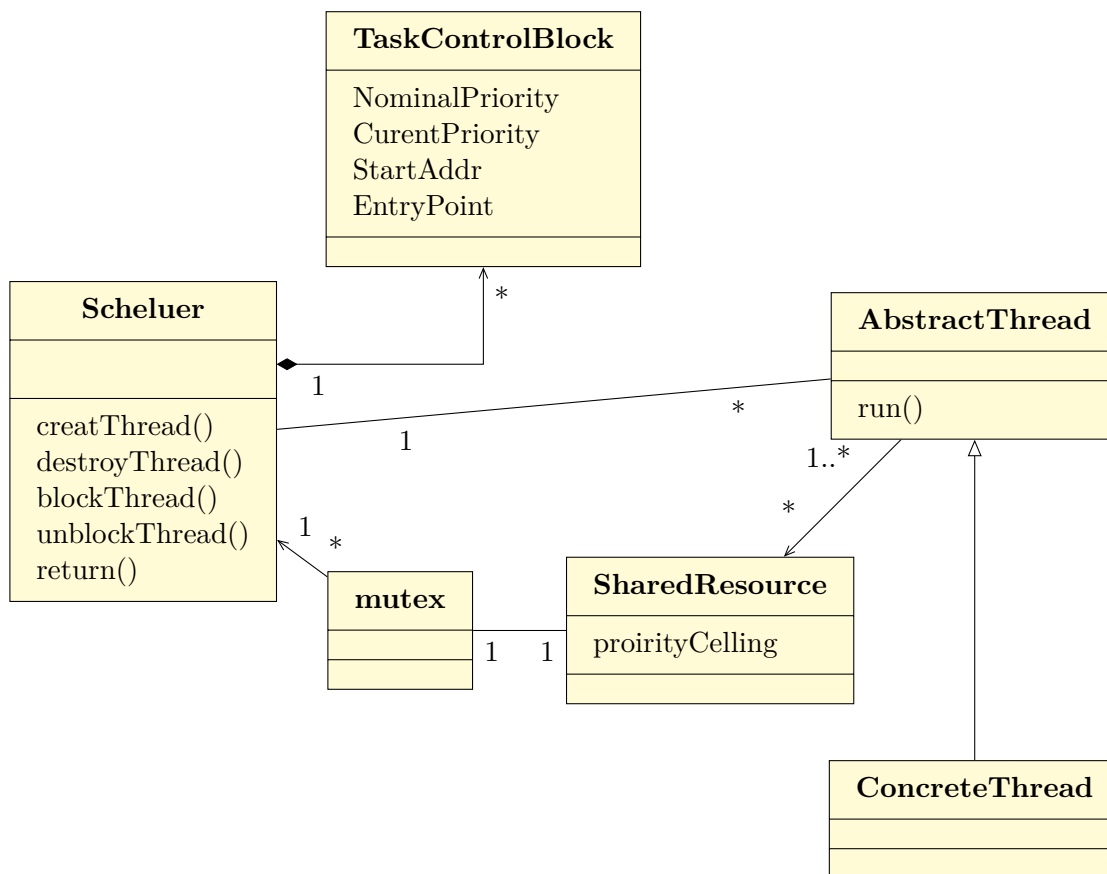
(این الگو همان **الگوی Simultaneous Locking** است که در [۱] بیان شده‌است).

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



شکل ۴۵: دیاگرام کلاس Highest Locker

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

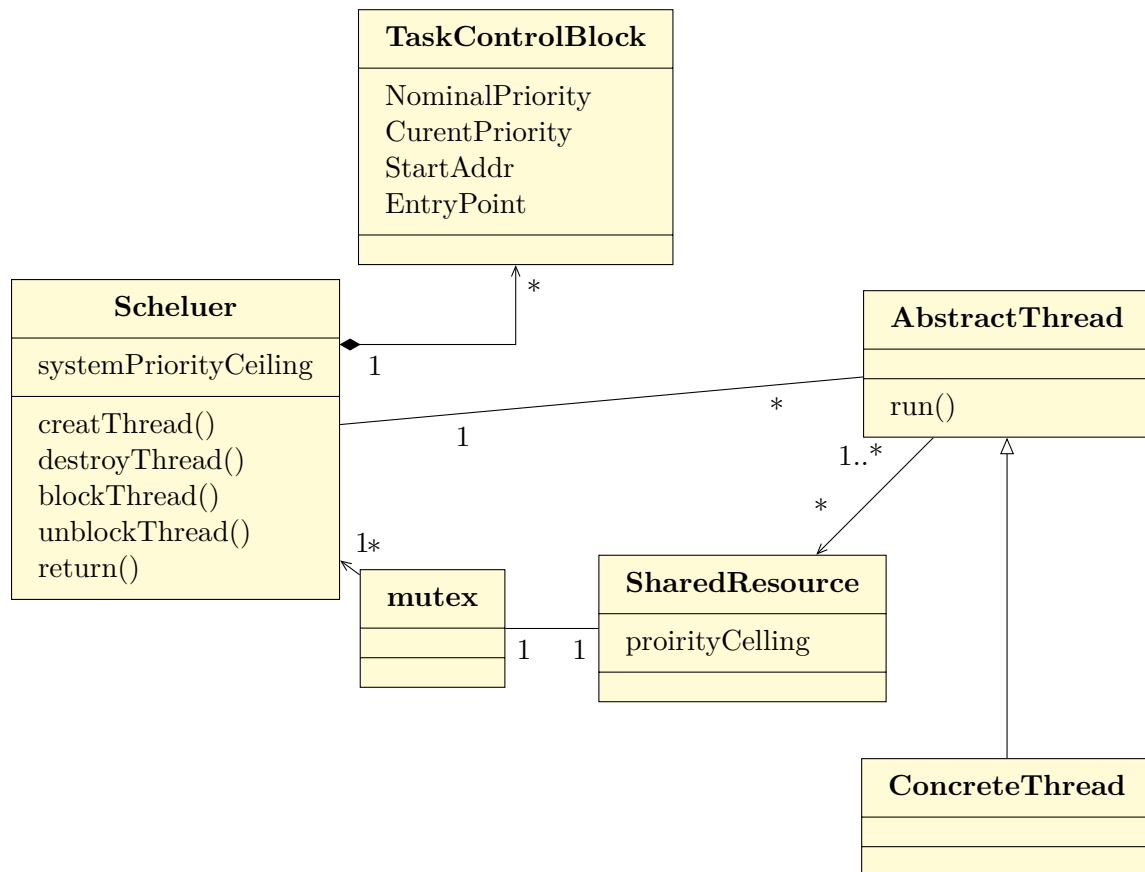
این الگو [۴] به پردازنده‌های متعدد امکان می‌دهد داده‌ها را با استفاده از یک ناحیه حافظه مشترک که معمولاً توسط چیپ‌های RAM چند پورت تسهیل می‌شود، به اشتراک بگذارند. این الگو ساده و مفید است وقتی که نیاز به اشتراک‌گذاری داده‌ها بین پردازنده‌ها وجود دارد، بدون نیاز به پاسخ‌های فوری به پیام‌ها یا رویدادها. 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



شکل ۴۶: دیاگرام کلاس Priority Ceiling

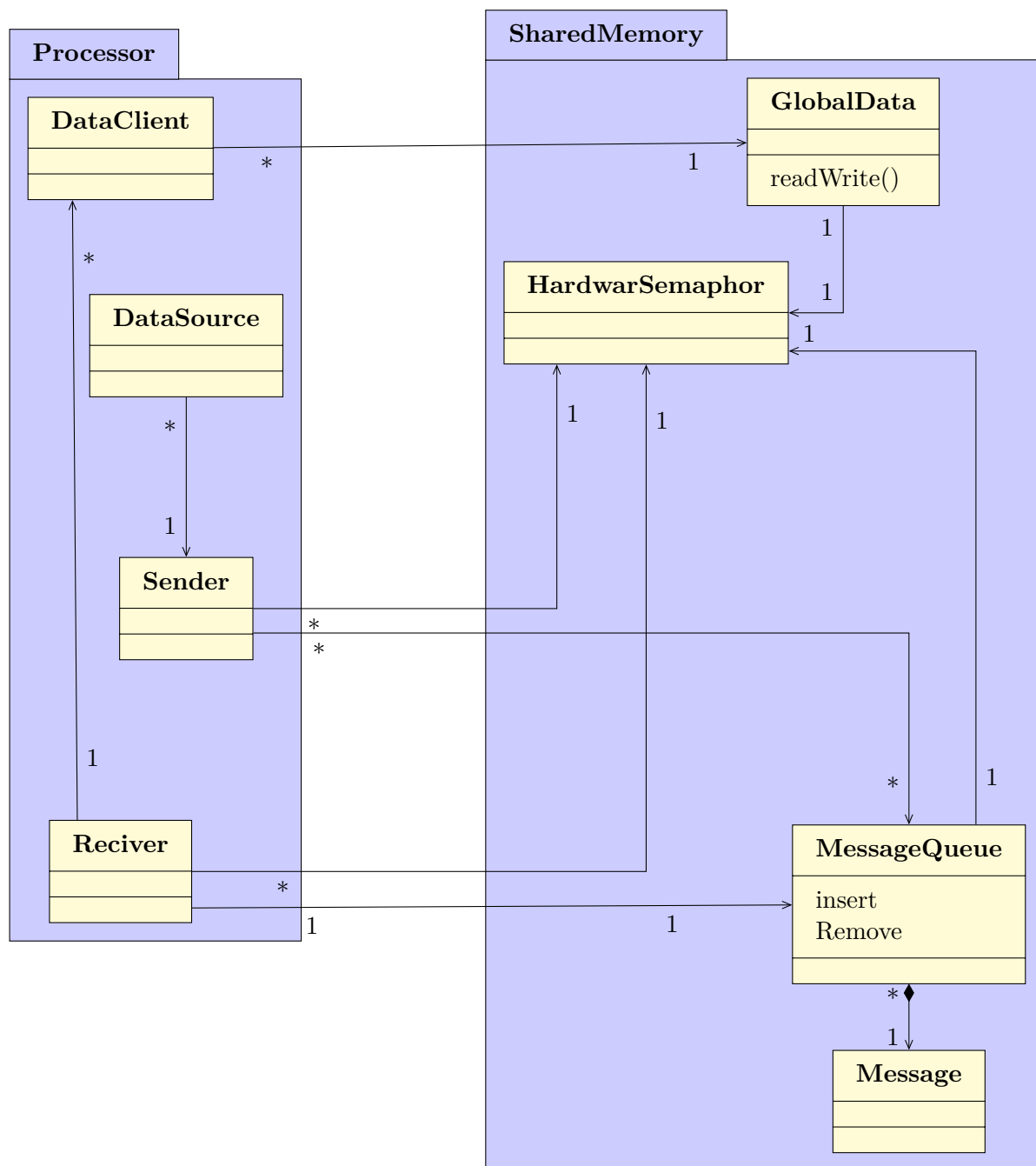
کلاینت-سرور را بر روی شبکه‌ها ساده‌تر می‌کنند. اگرچه تأخیرهای شبکه و پیچیدگی‌های مدیریت خطا ذاتی هستند، RMCها نسبت به IPC سنتی فرایند را ساده‌تر می‌کنند. انتخاب پروتکل انتقال مناسب، مانند TCP به جای UDP غیرقابل اعتماد، می‌تواند نگرانی‌های مربوط به به‌موقع‌بودن و قابلیت اطمینان را برطرف کند.

۳.۹.۲ الگوی Observer

این الگو همان الگوی Observer است که در [۱] گفته شده است؛ اما در این جا تعریف خود را به کاربردها ارتباط با سنسور محدود نکرده است و در ساختار سیستم‌های توزیع‌شده معنا پیدا می‌کند. در اینجا ساختار الگو کاملاً 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

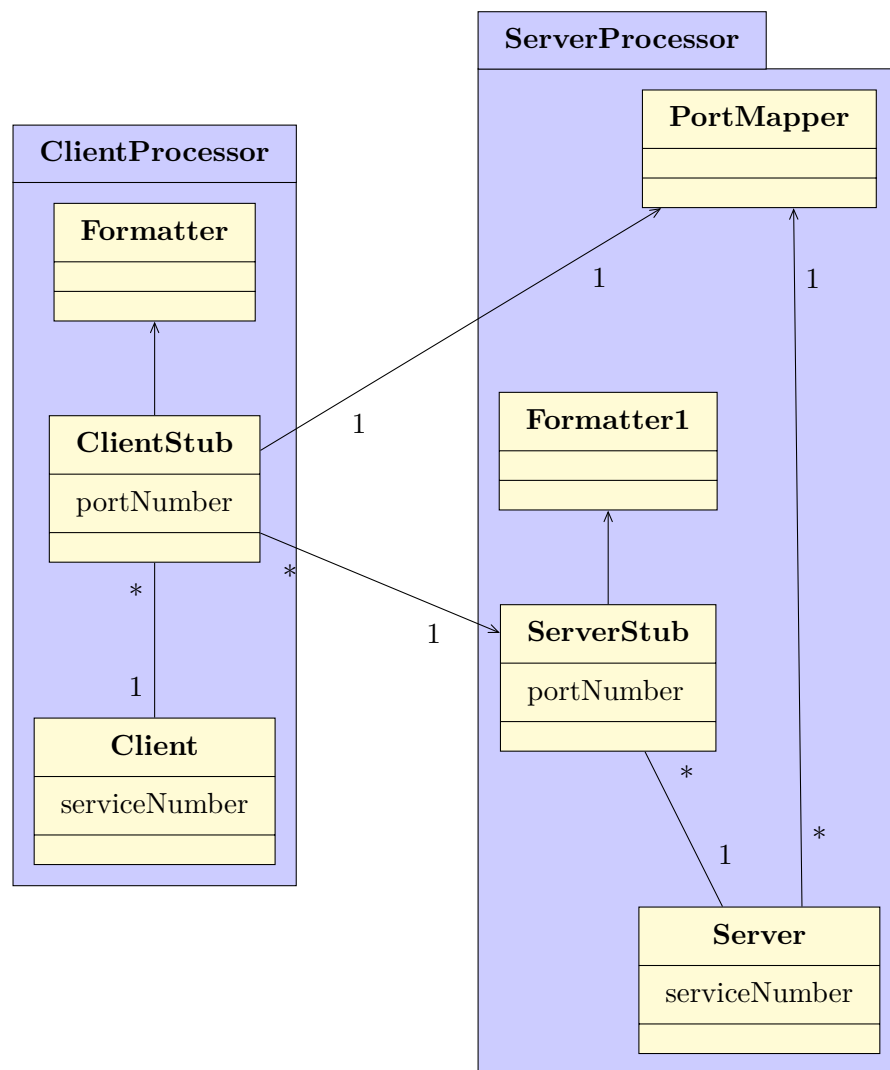
این الگو [۴] الگوی Observer را با ارائه یک گذرگاه مشترک گسترش می‌دهد که در آن چندین سرور اطلاعات خود را منتشر می‌کند و چندین کلاینت رویدادها و داده‌ها را دریافت می‌کنند. این الگو برای سیستم‌هایی که سرورها و کاربران زیادی باید داده‌ها را به اشتراک بگذارند مناسب است و توسط گذرگاه‌های سخت‌افزاری مانند گذرگاه 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 دریافت کنند 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.



شکل ۴۷: دیاگرام کلاس Shared Memory

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.



شکل ۴۸: دیاگرام کلاس Remote Method Call

۶.۹.۲ الگوی Broker

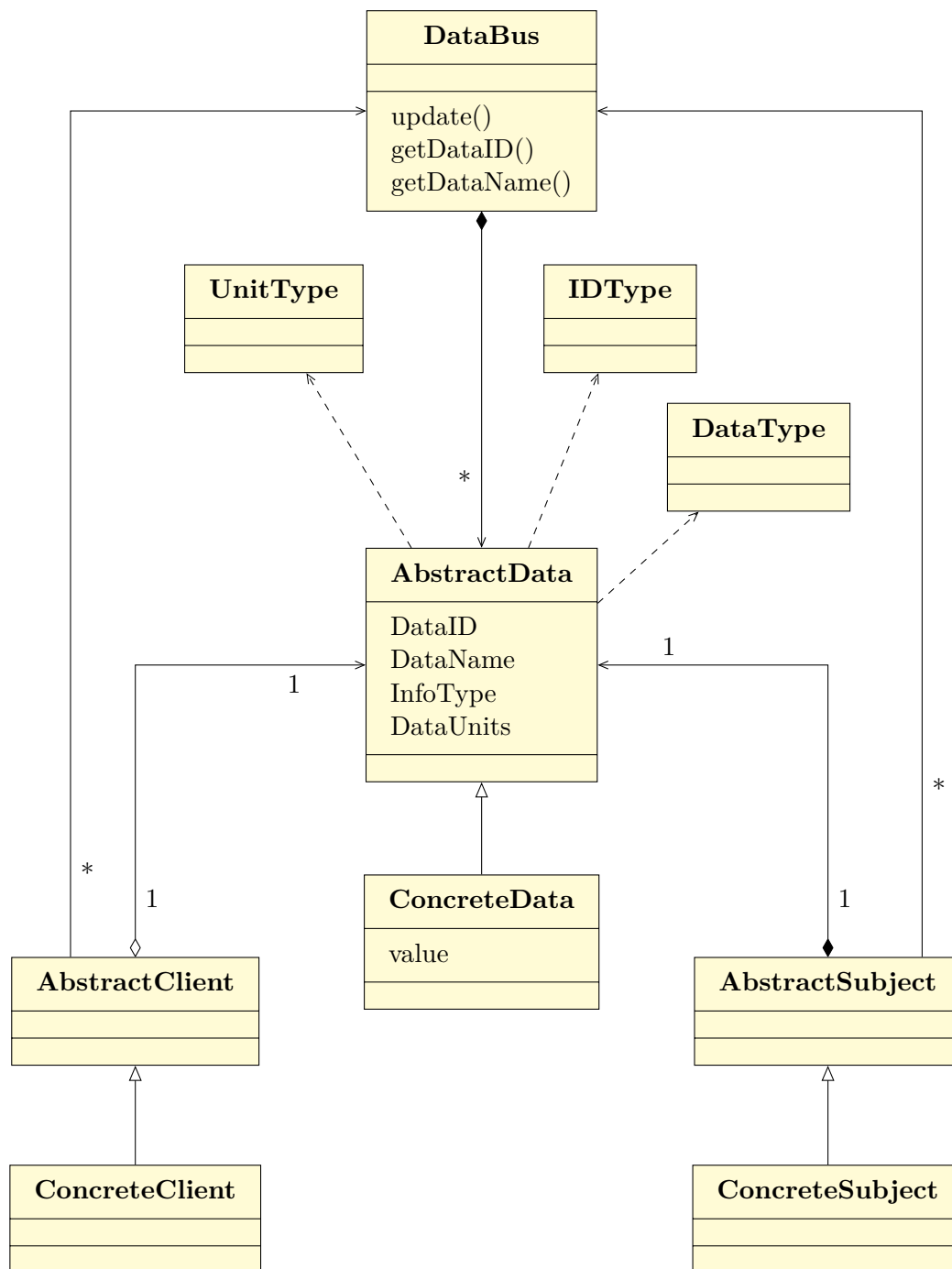
الگوی بروکر [۴] یک نسخه متقارن از الگوی Proxy است که برای شرایطی طراحی شده که مکان کلاینت و سرورها در زمان طراحی مشخص نیست. این الگو یک بروکر را معرفی می‌کند، که یک مخزن ارجاع شیء است و برای هر دو کلاینت و سرورها قابل مشاهده است و به کلاینت‌ها در یافتن سرورها کمک می‌کند. این کار، امکان استقرار معماری‌های 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 Broker (ORB) های 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 ORB های کوچکتر یا پیاده‌سازی سفارشی استفاده کرد.

۱۰.۲ الگوهای معماری امنیت و قابلیت اطمینان

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

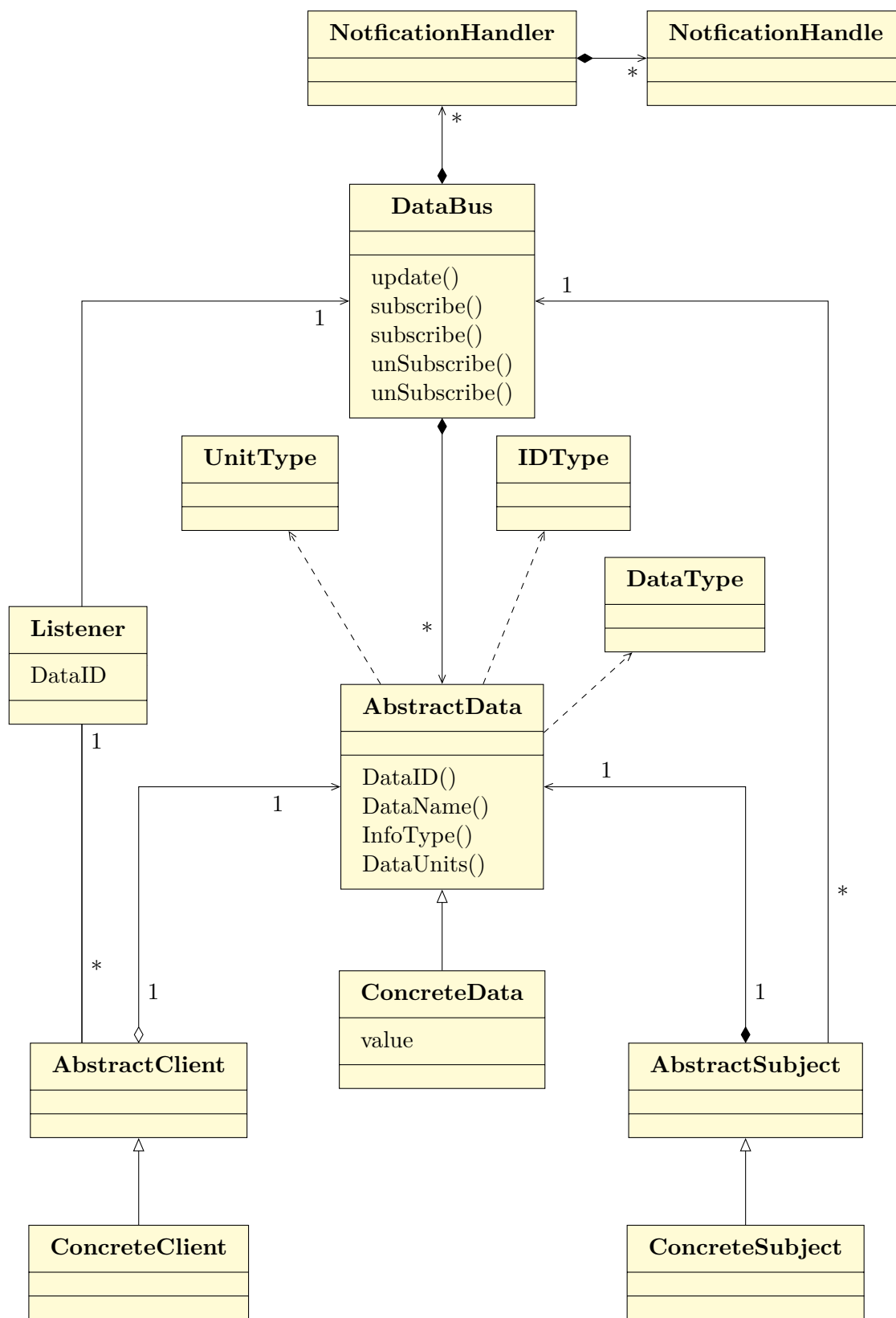
۱.۱۰.۲ الگوی Protected Single Channel

redundancy کامل در سیستم‌هایی که امنیت در آن‌ها حیاتی است، پرهزینه است، هم در تکرار سخت‌افزار و هم در توسعه آن. این الگو

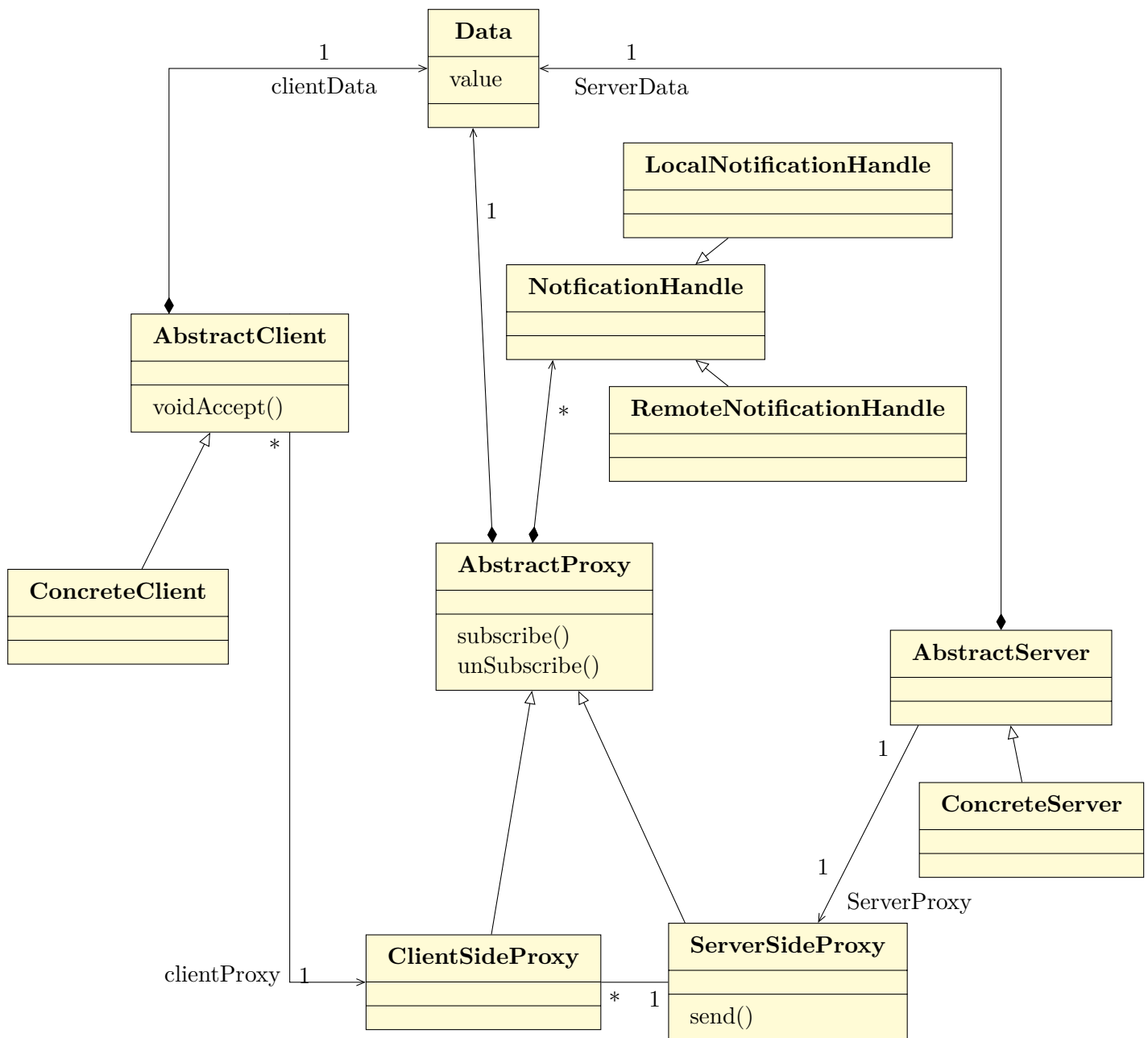


شکل ۴۹: دیاگرام کلاس (Data Bus (Pull Version))

[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 شده است؛ با این تفاوت که در اینجا در ابعاد معماری تعریف شده.



شکل ۵۰: دیاگرام کلاس Data Bus (Push Version)



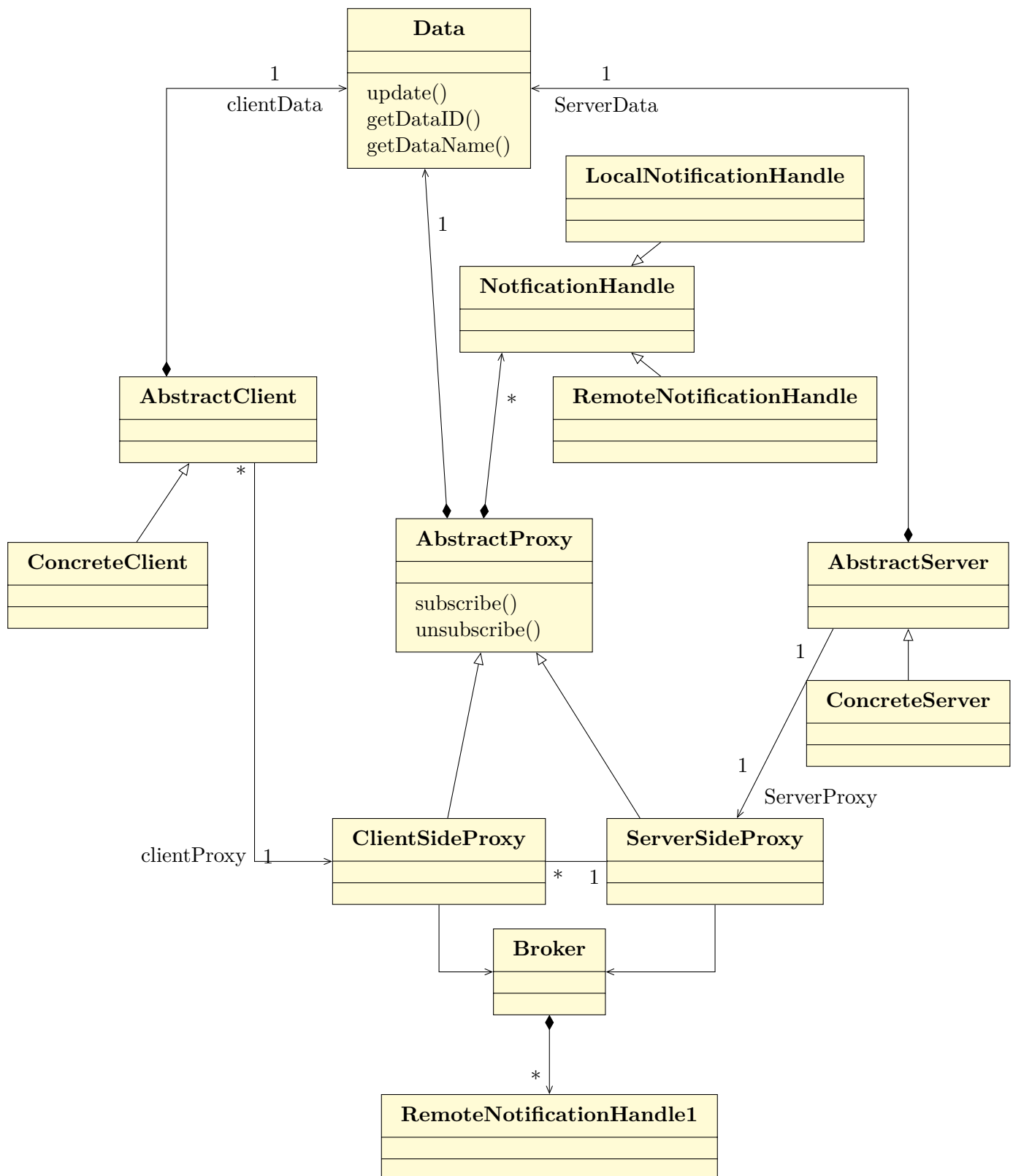
شکل ۵۱: دیاگرام کلاس Proxy

۲.۱.۰.۲ 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.

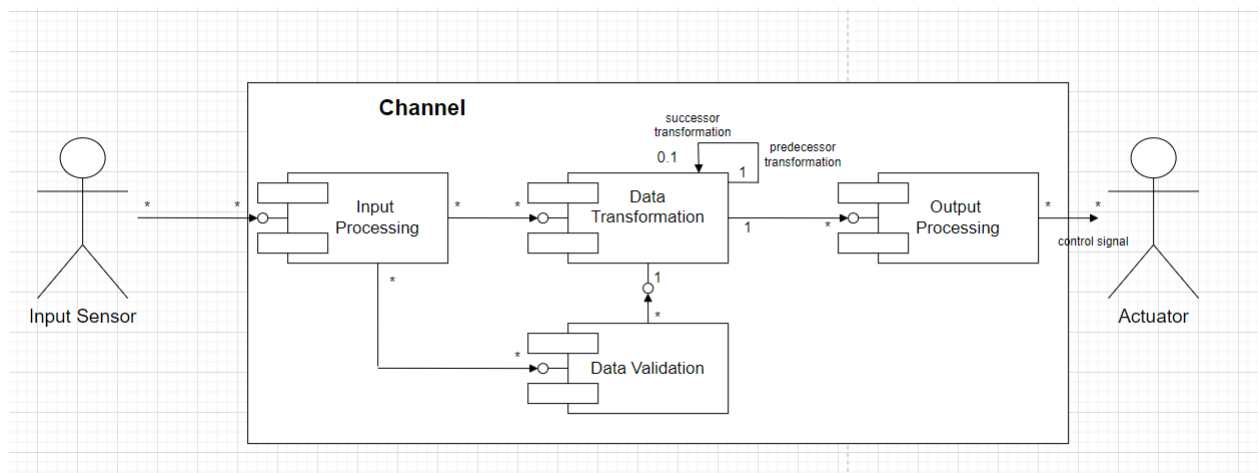
۳.۱.۰.۲ الگوی Triple Modular Redundancy

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



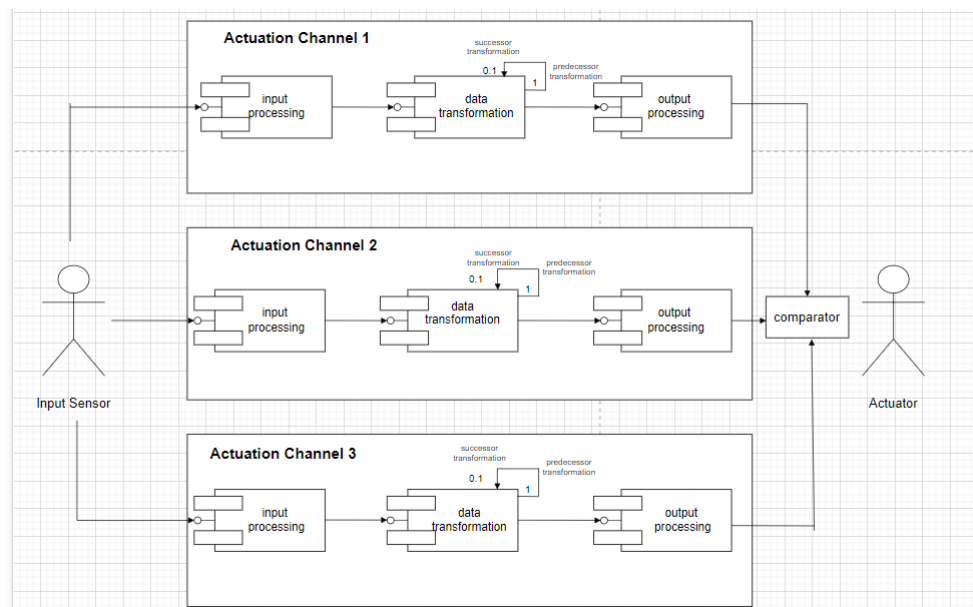
شکل ۵۲: دیاگرام کلاس Broker

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،



شکل ۵۳: ساختار الگوی Protected Single Channel

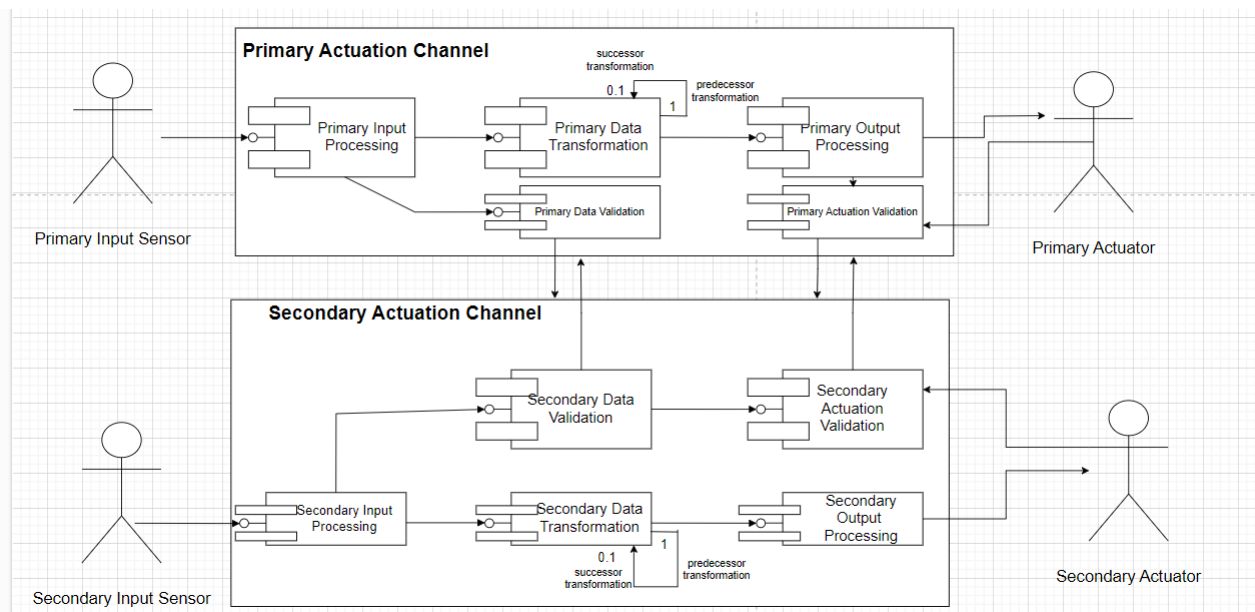
بدون استفاده از کانال‌های ناهمگن، از خطاهای سیستماتیک محافظت نمی‌کند. TMR به دلیل تکرار سخت‌افزار هزینه بالایی دارد، اما برای برنامه‌های بسیار حیاتی با نیاز به قابلیت اطمینان بالا و بدون وضعیت ایمن ضروری است.



شکل ۵۴: ساختار الگوی Triple Modular Redundancy

۴.۱.۰.۲ الگوی Heterogeneous Redundancy

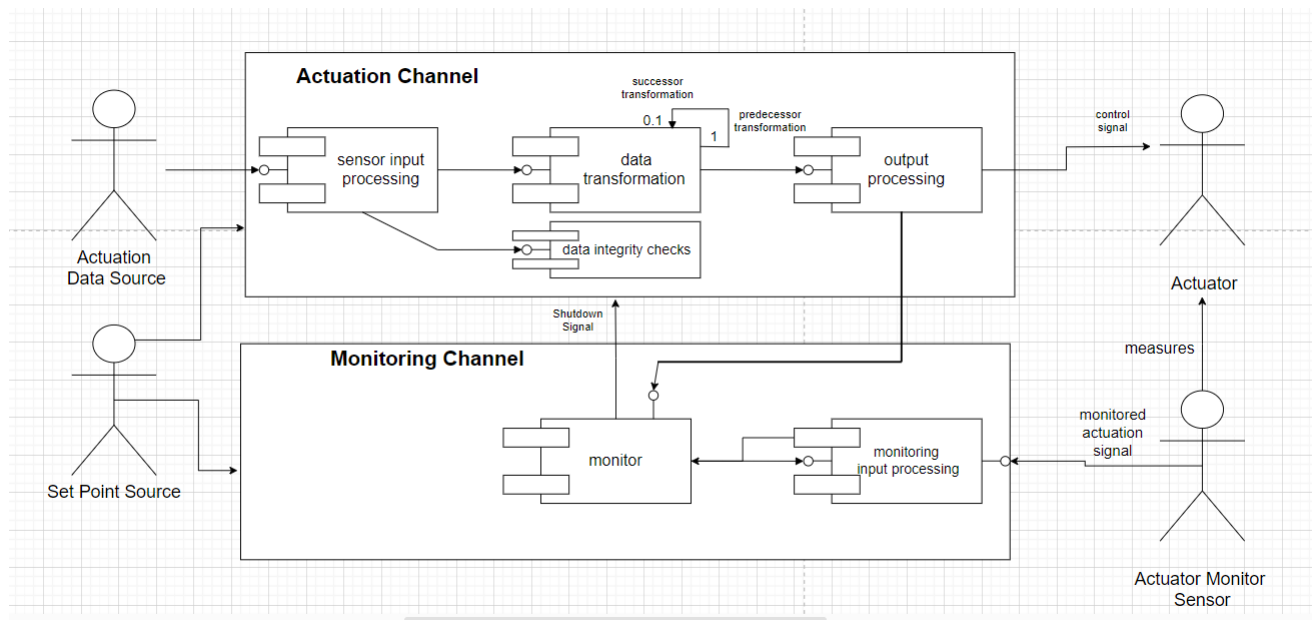
الگوی تکرار ناهمگن [۴] با استفاده از چندین کانال که به‌طور مستقل طراحی شده‌اند، تشخیص خطاها را بهبود می‌بخشد و هم خطاهای سیستماتیک و هم خطاهای تصادفی را شناسایی می‌کند. این الگو به دلیل نیاز بیشتر به زحمت توسعه و هزینه‌های تکراری، پرهزینه‌تر از الگوی **Homogeneous Redundancy** است. برای سیستم‌های با 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.



شکل ۵۵: ساختار الگوی Heterogeneous Redundancy

۵.۱۰.۲ الگوی Monitor-Actuator

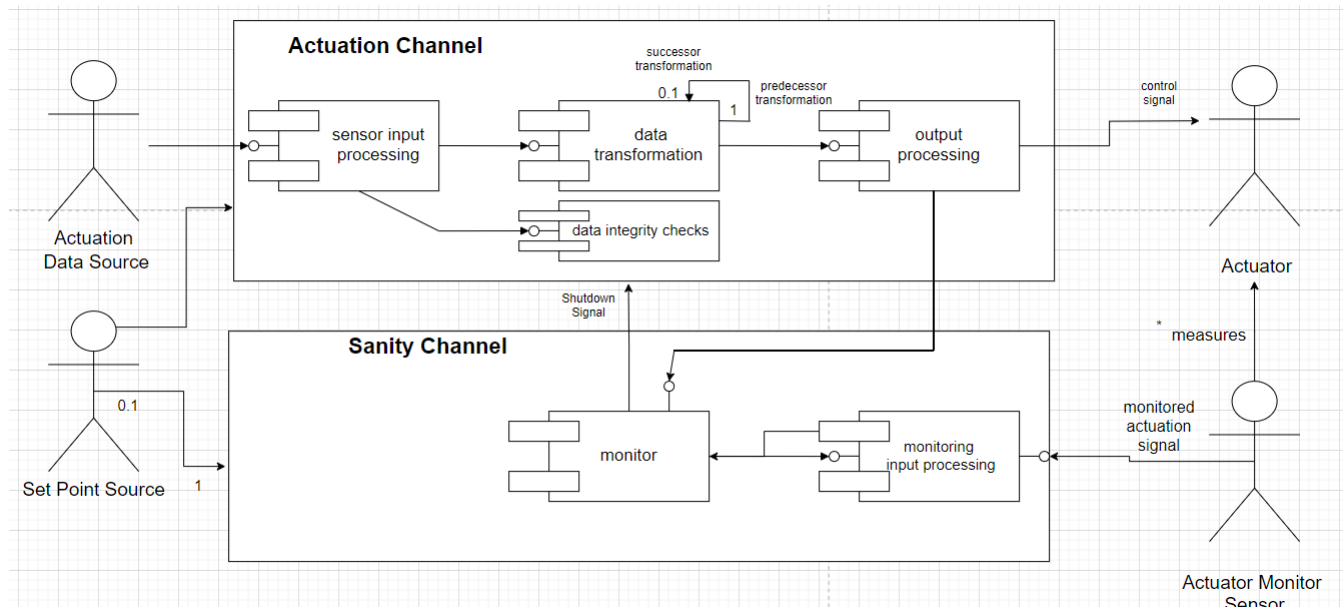
این الگو [۴] یک راهکار ایمنی مقرون به صرفه است که در سیستم‌هایی با نیازمندی‌های دسترسی متوسط تا پایین و حالت ایمن تعریف شده استفاده می‌شود. این الگو شامل یک حسگر مستقل است که کانال فعال‌سازی را برای شناسایی خطاها نظارت می‌کند و اطمینان می‌دهد که سیستم در صورت لزوم به حالت ایمن وارد می‌شود. این الگو شکل 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.



شکل ۵۶: ساختار الگوی Monitor Actuator

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 نمی‌بیند. 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.



شکل ۵۷: ساختار الگوی Sanity Check

۷.۱۰.۲ الگوی Watchdog

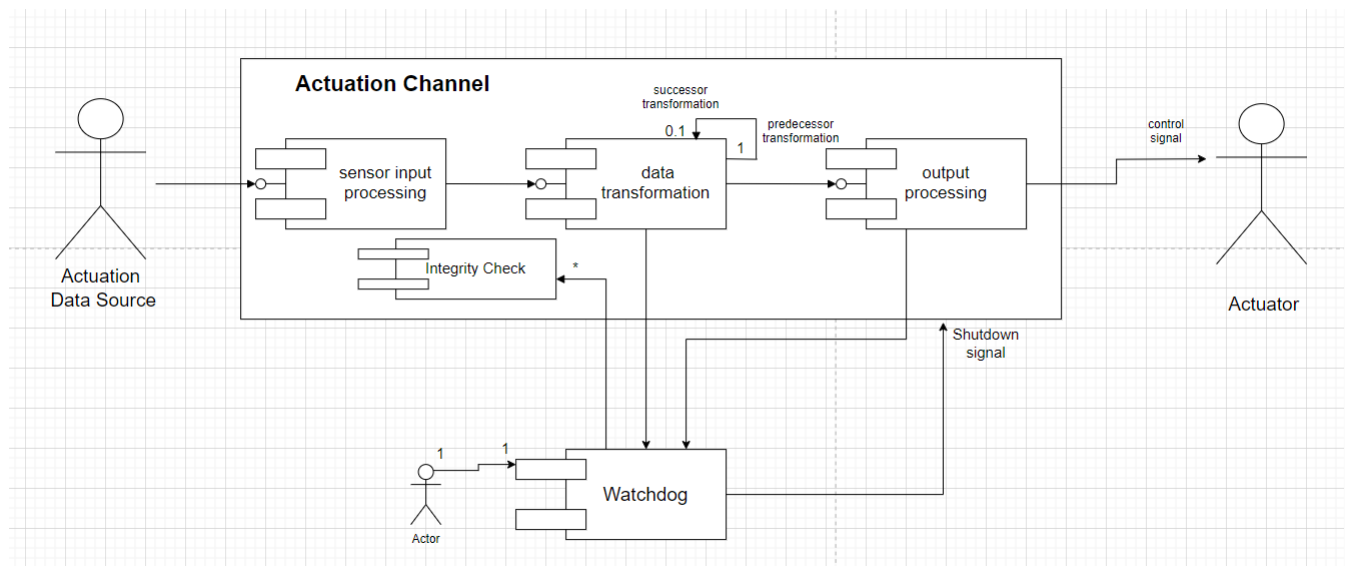
این الگو [۴] یک روش سبک و کم‌هزینه برای اطمینان از عملکرد صحیح فرآیندهای محاسبات داخلی است. برخلاف الگوی **Sanity Check** که خروجی سیستم را با استفاده از حسگرهای خارجی نظارت می‌کند، الگوی **Watchdog** بررسی می‌کند که محاسبات به درستی و به موقع انجام شوند. این الگو پوشش خطای حداقلی ارائه می‌دهد و عمدتاً 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

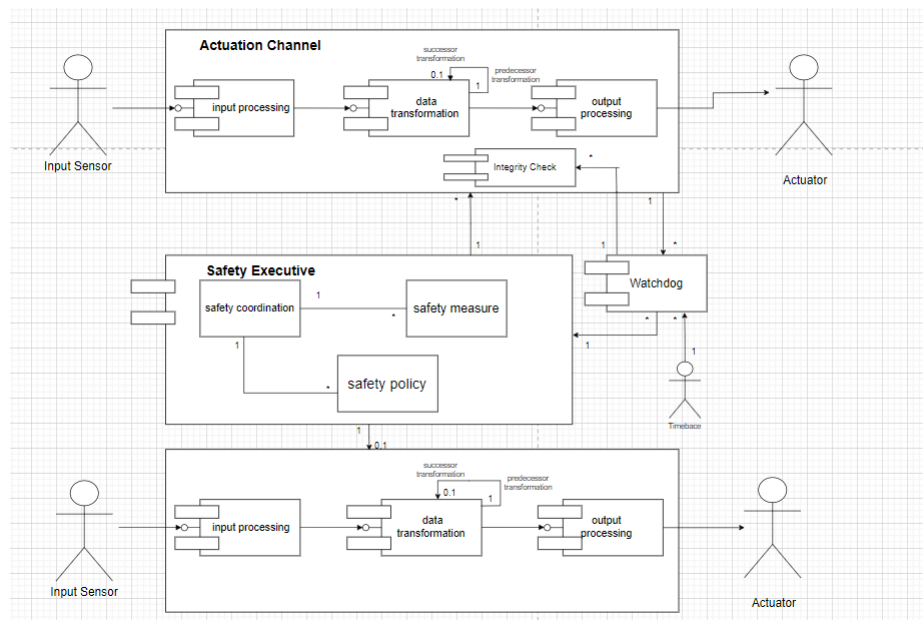
این الگو [۴] برای سیستم‌هایی طراحی شده است که اقدامات ایمنی پیچیده‌ای دارند و نمی‌توان آنها را به سادگی با خاموش کردن سیستم به دست آورد. این الگو یک جزء مجری ایمنی معرفی می‌کند تا چندین کانال و اقدامات ایمنی را مدیریت و هماهنگ کند و سیستم را از طریق یک سری مراحل به وضعیت ایمن هدایت کند. این الگو به‌ویژه 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.



شکل ۵۸: ساختار الگوی Watchdog



شکل ۵۹: ساختار الگوی Safety Executive

۱.۱۱.۲ الگوی Homogeneous Duplex

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

۲.۱۱.۲ الگوی Heterogeneous Duplex

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

۳.۱۱.۲ الگوی Triple Modular Redundancy

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

۴.۱۱.۲ الگوی M-Out-Of-N

الگوی M out of N [۵] با ارائه تکرار همگن، شامل N ماژول یکسان است که به طور موازی برای مخفی کردن خطاهای تصادفی و افزایش ایمنی و قابلیت اطمینان سیستم کار می کنند. این الگو نیاز دارد که حداقل M از N ماژول برای عملکرد سیستم موفق شوند و از یک الگوریتم رأی گیری برای مدیریت خطاهای تصادفی بدون از دست دادن داده های ورودی استفاده می کند. این الگو برای سیستم هایی با نرخ بالای خطاهای تصادفی و بدون محدودیت تکرار مناسب است. تأثیر کمی بر قابلیت تغییر سیستم یا زمان اجرا دارد، اما در برابر خطاهای سیستماتیک بی اثر است، زیرا کانال های یکسانی دارند که دارای همان خطاهای احتمالی هستند. استفاده از چندین حسگر برای جلوگیری از خرابی نقطه ای واحد ممکن است به مشکلاتی منجر شود، به ویژه به دلیل تفاوت در زمان پاسخ حسگرها.

۵.۱۱.۲ الگوی Monitor-Actuator

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

۶.۱۱.۲ الگوی Sanity Check

(این الگو همان الگوی [Sanity Check](#) است که در [۴] نوشته شده.)

۷.۱۱.۲ الگوی Watchdog

(این الگو همان الگوی [Watchdog](#) است که در [۴] نوشته شده.)

۸.۱۱.۲ الگوی Safety Executive

(این الگو همان الگوی [Safety Executive](#) است که در [۴] نوشته شده.)

۱۲.۲ الگوهای نرم افزاری برای سیستم های Safety-Critical

این دسته از الگوها که در [۵] بیان شده اند، با استفاده از ساختارهای تکراری در نرم افزار می خواهند شرایط قابلیت اطمینان و امنیت را ایجاد کنند.

۱.۱۲.۲ الگوی N-Version Programming

الگوی (N-Version Programming (NVP [۵] یک روش نرم افزاری تحمل خطا است که بر تنوع نرم افزار و مخفی سازی خطاها متکی است. این روش شامل ایجاد N نسخه نرم افزاری معادل عملکردی ($N \geq 2$) به صورت مستقل از مشخصات اولیه است. این نسخه ها به طور موازی اجرا می شوند و وظیفه یکسانی را با ورودی یکسان انجام می دهند تا N خروجی تولید کنند. در این الگو، یک رأی گیر برای تعیین خروجی صحیح با استفاده از نتایج N نسخه به کار می رود. NVP برای سیستم های با ایمنی بسیار بالا مناسب است، زمانی که نیاز به نرم افزار بسیار قابل اعتماد وجود دارد، هزینه بالای توسعه نسخه های متعدد قابل تحمل است، تیم های مستقل برای توسعه نسخه های مختلف موجود هستند و واحدهای سخت افزاری تکراری برای اجرای این نسخه ها به صورت موازی قابل استفاده هستند. با این حال، NVP دارای معایبی مانند پیچیدگی و هزینه بالای توسعه نسخه های مستقل و وابستگی به مشخصات اولیه است که می تواند خطاها را به تمامی نسخه ها منتقل کند و ایمنی و قابلیت اطمینان سیستم را تحت تأثیر قرار دهد.

۲.۱۲.۲ الگوی Recovery Block

این الگو [۵] یک روش نرم افزاری تحمل خطا است که از تشخیص خطا با آزمون های پذیرش و بازیابی اطلاعات برای جلوگیری از خرابی سیستم استفاده می کند. مشابه الگوی [N-Version Programming](#)، این روش شامل ایجاد N نسخه نرم افزاری متنوع، مستقل و معادل عملکردی از مشخصات اولیه است. این نسخه ها به نسخه اصلی و $N-1$ نسخه ثانویه تقسیم می شوند. اجرای هر نسخه با یک آزمون پذیرش دنبال می شود. اگر نسخه اصلی در آزمون خود شکست بخورد، یک نسخه ثانویه Recovery Block اجرا می شود و پس از آن آزمون پذیرش انجام می شود. این فرآیند تا زمانی که یک نسخه آزمون پذیرش را بگذارد یا همه نسخه ها شکست بخورند و یک خرابی کلی سیستم گزارش شود، تکرار می شود. این روش برای سیستم های با ایمنی بسیار بالا مناسب است، زمانی که نرم افزار بسیار قابل اعتماد و ایمنی نیاز است، امکان ساخت آزمون پذیرش برای اطمینان از عملکرد صحیح نرم افزار و تشخیص خروجی های ممکن نادرست وجود دارد، تیم های مستقل برای توسعه نسخه های مختلف موجود هستند و هزینه بالای توسعه نسخه های 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 [۵] (AVP) 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 ($N \geq 2$) 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** گفته شد، 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 اجزای 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 بگذرانند. 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] آمده است.

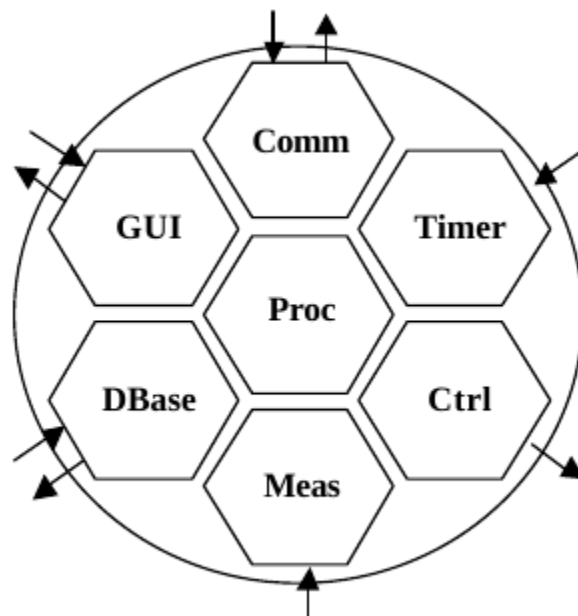
۲.۱۳.۲ الگوی 3-Level Safety Monitoring

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 Acuatorها 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.

۱۴.۲ فریم‌ورک ساخت الگوهای معماری

Zalewski در [۲] یک معماری برای سیستم‌های نهفته بی‌درنگ معرفی می‌کند که آن قدر کلی بیان شده که از آن می‌توان به عنوان یک فریم‌ورک برای ساخت الگوهای جدید استفاده کرد. او ساختار خود را مبنی بر ارتباط سیستم‌های بی‌درنگ با دنیای بیرون از خود می‌سازد. ساختار پیشنهادی او در شکل ۶۰ نمایش داده شده.



شکل ۶۰: معماری پیشنهادی Zalewski [۲]

در این ساختار، Comm برای ارتباط با دیگر ساختارهای کنترلی با استفاده از یک شبکه ارتباطی، GUI برای ارتباط با کاربر سیستم، DBase برای ارتباط با دیتابیس، Meas برای ارتباط با سنسورها، Ctrl برای کنترل و Actuate کردن بخش‌های سیستم، Timer برای ارتباط با سیستم بی‌درنگ و Proc برای انجام پردازش‌های درونی سیستم تعریف شده‌اند.

۳ تحلیل

همانطور که در بخش پیشینه پژوهش دیدیم، الگوهای گفته شده را می توان به دو دسته اصلی الگوهای معماری و الگوهای طراحی تقسیم بندی کرد.

۱.۳ الگوهای معماری

با توجه به الگوهایی که به عنوان الگوی معماری در ادامه می آوریم، الگوها را به ۶ دسته تقسیم می کنیم.

۱. From Mud to Structure

۲. Distributed Systems

۳. Interactive Systems

۴. Adaptable Systems

۵. Safe Channel-based Systems

۶. Safe Systems without Channel

همانطور که در بالا دیده می شود، چهار دسته اول تقسیم بندی را براساس [۶] آورده و دو دسته بعدی را این گزارش معرفی می کند. در ادامه الگوهای گفته شده در بخش پیشینه پژوهشی را مورد بررسی قرار داده و در این دسته بندی قرار می دهیم. گفتنی است که تعدادی از الگوهایی که به عنوان الگوی معماری در بخش پیشینه پژوهشی آورده شده اند، با توجه به توضیحاتی که درباره هر کدام می دهیم جزو الگوهای طراحی باید طبقه بندی شوند.

۱.۱.۳ From Mud to Structure

□ **الگوی Layered**: با توجه به تعریف ارائه شده، این الگو با بیان ساختار یک سیستم و نحوه ارتباط زیربخش های آن با هم، در دسته الگوهای معماری قرار می گیرد. لازم به ذکر است که این الگو همان الگوی Layers است که Buschmann و غیره [۶] گفته اند.

□ **الگوی Five Layer**: این الگو نیز یکی از حالت های خاص الگوی Layered است و در همین دسته قرار می گیرد.

□ **الگوی Channel**: این الگو در [۴] به عنوان یک الگوی معماری و در [۱] به عنوان یک الگوی طراحی معرفی شده است. طبق تعریف، این الگو با استفاده از یک زنجیره از اشیا می تواند توابعی را روی داده، از ابتدا (Source) تا انتها (Sink) انجام دهد. این رفتار بسیار شبیه الگوی Pipes and Filters در [۶] است. در این الگو اگر کانال یک عنصر درشت دانه باشد و جزوی از یک زیربخش قابل تکرار در سیستم باشد، می توان این الگو را در دسته الگوهای معماری قرار داد.

□ **الگوی Recursive Containment**: این الگو با تعریف یک زیربخش سیستمی به عنوان عنصر اصلی، می گوید که زیربخش ها همگی با صورت تودرتو می توانند درون یکدیگر قرار بگیرند و سیستم را تشکیل دهند. این الگو به دلیل درشت دانه بودن ساختار اجزای تشکیل دهنده، در این دسته قرار می گیرد.

□ **الگوی Hierarchical Control**: این الگو نیز شباهت زیادی با الگوی Recursive Containment داشته و در این دسته قرار می گیرد.

۲.۱.۳ Distributed Systems

□ **الگوی Shared Memory**: این الگو با به اشتراک گذاری یک واحد حافظه بین چندین پردازنده، امکان استفاده پردازنده ها از یک حافظه مشترک را فراهم می سازد. اجزای تشکیل دهنده این الگو، پردازنده ها و حافظه ها هستند که اجزای درشت دانه سیستم های نهفته به شمار می روند. از این رو این الگو در دسته الگوهای معماری قرار می گیرد.

□ **الگوی Remote Method Call**: این الگو با فراهم کردن یک پروتکل ارتباطی بین کلاینت و سرور، در یک ساختار توزیع شده اجازه فراخوانی توابع سرور توسط کلاینت را فراهم می سازد. از این رو این الگو در دسته الگوهای معماری قرار می گیرد.

□ **الگوی Data Bus**: این الگو با ایجاد یک گذرگاه داده، اجازه می‌دهد زیربخش‌های یک سیستم با استفاده از آن، با یکدیگر ارتباط داشته باشند و بتوانند روی این گذرگاه، داده‌های مورد نیاز را ارسال و دریافت کنند. این الگو در سیستم‌های نهفته کاربردهای زیادی دارد و معمولاً بین سخت‌افزارهای مجزا استفاده می‌شود. این الگو نیز جزو دسته الگوهای معماری است.

□ **الگوی Broker**: این الگو کاملاً مشابه الگوی Broker در [۶] است و در دسته الگوهای معماری قرار می‌گیرد.

۳.۱.۳ Interactive Systems

□ **فریم‌ورک ساخت الگوهای معماری**: این الگو با ایجاد یک ساختار معماری با در نظر داشتن رابط کاربری و ارتباط بخش‌های مختلف سیستم و سیستم‌های مختلف با هم، یک معماری قابل استفاده برای کاربران ایجاد می‌کند که استفاده از سیستم نهفته را ممکن می‌سازد.

۴.۱.۳ Adaptable Systems

□ **الگوی Microkernel**: این الگو نیز همان الگوی Microkernel است که در [۶] گفته شده.

□ **الگوی Virtual Machine**: این الگو یک سیستم کامل را بر روی یک ماشین مجازی پیاده‌سازی کرده و با هدف تطبیق‌پذیری بالا طراحی شده است. این الگو نحوه استقرار یک سیستم نرم‌افزاری بر روی پلتفرم‌های مختلف را بررسی می‌کند و مشخصاً یک الگوی معماری است.

□ **معماری Component-Based**: این الگو به ایجاد یک سیستم نرم‌افزاری مبتنی بر اجزای قابل استفاده مجدد تکیه دارد و یک الگوی معماری است.

۵.۱.۳ Safe Channel-based Systems

□ **الگوی Homogeneous Redundancy**: این الگو بر پایه الگوی الگوی Channel ساخته شده و صرفاً برای منظور امنیت، یک کانال اضافی نیز تعریف می‌کند.

□ **الگوی Triple Modular Redundancy**: مشابهاً در این الگو نیز از کانال به تعداد سه مرتبه استفاده شده است.

□ **الگوی Heterogeneous Redundancy**: این الگو بر پایه الگوی الگوی Channel ساخته شده و صرفاً برای منظور امنیت، یک کانال اضافی از نوعی دیگر نیز تعریف می‌کند.

□ **الگوی Monitor-Actuator**: این الگو نیز با استفاده از کانال‌ها یک ساختار معماری امن ارائه می‌کند.

□ **الگوی Sanity Check**: این الگو نیز با استفاده از کانال‌ها باعث ایجاد یک ساختار امن می‌شود.

□ **الگوی Watchdog**: این الگو نیز با استفاده از کانال‌ها باعث ایجاد یک ساختار امن می‌شود.

□ **الگوی Safety Executive**: این الگو نیز با استفاده از کانال‌ها باعث ایجاد یک ساختار امن می‌شود.

□ **الگوی M-Out-Of-N**: این الگو نسخه تعمیم‌یافته الگوی Triple Modular Redundancy است.

□ **الگوی 3-Level Safety Monitoring**: این الگو با استفاده از تکرار نرم‌افزار به علاوه استفاده از کانال‌ها، امنیت را فراهم می‌کند.

۶.۱.۳ Safe Systems without Channel

□ **الگوی N-Version Programming**: این الگو با ایجاد تکرار در نرم‌افزار، به امنیت و قابلیت اطمینان کمک می‌کند. با توجه به این که در این الگو، اجزا در سطح زیرسیستم هستند، می‌توان این الگو را در دسته الگوهای معماری در نظر گرفت.

□ **الگوی Recovery Block**: مشابه الگوی قبلی با تکرار نرم‌افزار امنیت را فراهم می‌کند.

□ **الگوی Acceptance Voting**: مشابه الگوی قبلی با تکرار نرم‌افزار امنیت را فراهم می‌کند.

□ **الگوی N-Self Checking Programming**: مشابه الگوی قبلی با تکرار نرم‌افزار امنیت را فراهم می‌کند.

□ **الگوی Recovery Block with Backup Voting**: مشابه الگوی قبلی با تکرار نرم‌افزار امنیت را فراهم می‌کند.

۲.۳ الگوهای طراحی

در این بخش، الگوها را به دسته‌های زیر تقسیم‌بندی می‌کنیم. این تقسیم‌بندی تاحدی از [۱] و [۴] برگرفته شده و تاحدی ساخته این گزارش است.

۱. Hardware Patterns

۲. Concurrent Scheduling Patterns

۳. Concurrent Resource Sharing Patterns

۴. State Machine Patterns

۵. Safety with Redundancy Patterns

۶. Safety without Redundancy Patterns

۷. Memory Patterns

۱.۲.۳ Hardware Patterns

□ **الگوی Hardware Proxy**: این الگو با تعریف یک Remote Proxy [۳] برای سخت‌افزار، کدهای کلاینت را از سخت‌افزار ایزوله می‌کند. این الگو از نوع ساختاری است.

□ **الگوی Hardware Adapter**: این الگو دقیقاً همان الگوی Adapter در [۳] است. این الگو از نوع ساختاری است.

□ **الگوی Mediator**: این الگو دقیقاً مشابه الگوی Mediator در [۳] است. این الگو از نوع رفتاری است.

□ **الگوی Observer**: این الگو دقیقاً همان الگوی Observer در [۳] است. این الگو از نوع رفتاری است.

□ **الگوی Debouncing**: این الگو با برای استفاده از رابط‌های سخت‌افزاری که دچار مشکل Bouncing هستند، بسیار مفید است و به دلیل ایجاد رفتار جدید در ساختار خود و استفاده از یک سیستم تعلیق، جزو دسته الگوهای رفتاری [۳] قرار می‌گیرد.

□ **الگوی Interrupt**: این الگو در [۱] و [۴] در دو فرم الگوی طراحی و معماری بیان شده‌است. همانطور که می‌دانیم، وقفه‌ها در سیستم‌های نهفته بی‌درنگ، بخش مهمی از سیستم به شمار می‌روند که توسط سخت‌افزار در اختیار نرم‌افزار قرار می‌گیرند. الگوی Interrupt تاحدی شبیه الگوی Observer عمل می‌کند و کمک می‌کند که رویدادها را به صورت Non-Blocking دریافت کنیم. المان‌های حاضر در این الگو در سطح کلاس‌ها هستند و نه کامپوننت‌ها، لذا این الگو باید در دسته الگوهای طراحی قرار بگیرد نه الگوهای معماری. لازم به ذکر است که این الگو یک الگوی رفتاری است و فرایند اصلی اجرای برنامه را تحت تاثیر قرار می‌دهد.

□ **الگوی Polling**: این الگو هدفی مشابه هدف الگوی Interrupt دنبال می‌کند و تفاوت آن استفاده از ساختار Blocking است. این الگو نیز از دسته الگوهای رفتاری است.

۲.۲.۳ Concurrent Scheduling Patterns

□ **الگوی Cyclic Executive**: این الگو فلوای اجرای Threadها را کنترل می‌کند و وظیفه برنامه‌ریزی آن‌ها را انجام می‌دهد. این الگو از دسته الگوهای رفتاری است. این الگو در [۴] به عنوان یک الگوی معماری نیز معرفی شده‌است که درست نیست. زیرا که این الگو از اجزای ریزدانه ساخته شده و هدف کوچکی از یک سیستم نهفته بی‌درنگ را محقق می‌سازد.

□ **الگوی Static Priority**: این الگو مشابه الگوی Cyclic Executive برنامه ریزی Threadها را انجام می‌دهد؛ البته به شکلی متفاوت. این الگو نیز از نوع الگوهای رفتاری است. این الگو نیز در [۴] به عنوان یک الگوی معماری آمده‌است که به دلیل مشابه با الگوی Cyclic Executive باید در دسته الگوهای طراحی قرار بگیرد.

□ **الگوی Dynamic Priority**: این الگو هدفی مشابه الگوی Static Priority دنبال می‌کند و مانند آن، یک الگوی طراحی است.

- **الگوی Critical Region**: این الگو با هدف ایجاد اطمینان از انجام یک Thread خاص و عدم ایجاد مزاحمت برای آن ایجاد شده. این الگو از دسته الگوهای رفتاری است.
- **الگوی Round Robin**: هدف این الگو مانند هدف الگوی Cyclic Executive است و به دلایل مشابهی، یک الگوی طراحی است. این الگو از نوع الگوهای رفتاری است.

۳.۲.۳ Concurrent Resource Sharing Patterns

- **الگوی Guarded Call**: این الگو با هدف محافظت از یک منبع خاص در مقابل Race بین Threadها محافظت می‌کند. این الگو با اضافه کردن یک Semaphore به ساختار، این کار را انجام می‌دهد و یک الگوی رفتاری است. این الگو در [۴] به عنوان یک الگوی معماری بیان شده است و می‌گوید که اگر یک کلاینت بخواهد یک منبع از سرور را فراخوانی کند، این الگوی معماری مورد استفاده قرار می‌گیرد. در صورتی که این حرف دقیق نیست. چیزی که در شکل ۳۳ می‌بینیم، در اصل ترکیب الگوی Remote Method Call و الگوی Guarded Call است. که الگوی اول یک الگوی معماری و الگوی دوم (همین الگو) یک الگوی طراحی است.
- **الگوی Queuing**: این الگو با ایجاد یک سیستم مبتنی بر FIFO، یک سیستم امن برای انتقال داده بین Threadها ایجاد می‌کند. این الگو نیز به مدیریت یک منبع مشترک بین Threadها مشغول است و یک الگوی رفتاری است.
- **الگوی Rendezvous**: این الگو نیز با ایجاد یک کلاس مجزا برای سنکرون سازی Threadها، دسترسی به منابع را برای آنها به صورت امن فراهم می‌کند. این الگو نیز یک الگوی رفتاری است زیرا که کلاس Rendezvous فرایندهای سنکرون سازی را پیاده سازی کرده است. این الگو در [۴] به عنوان یک الگوی معماری نیز بیان شده است. اما این الگو به اندازه کافی کلی نیست که بخواهد در دسته الگوهای معماری باشد. همچنین ساختار این الگو بر اساس کلاسها است و نه کامپوننتها و زیرسیستمها.
- **الگوی Simultaneous Locking**: این الگو نیز با معرفی کلاسهای جدید، فرایند دسترسی امن به منابع مشترک را به نحو دیگری پیاده سازی می‌کند و یک الگوی رفتاری است.
- **الگوی Ordered Locking**: هدف این الگو نیز مانند الگوی Simultaneous Locking است و یک الگوی رفتاری است.
- **الگوی Priority Inheritance**: این الگو هدفی مانند الگوی Static Priority دارد و یک الگوی رفتاری است.
- **الگوی Highest Locker**: این الگو هدفی مانند الگوی Priority Inheritance دارد و یک الگوی رفتاری است.
- **الگوی Priority Ceiling**: این الگو هدفی مانند الگوی Priority Inheritance دارد و یک الگوی رفتاری است.

۴.۲.۳ State Machine Patterns

- **الگوی Single Event Receptor**: در این الگو، تنها یک شیء دریافت کننده رویدادها از بیرون است و آنها را به ماشین حالت پاس می‌دهد. این الگو به صورت یک الگوی رفتاری است.
- **الگوی Multiple Event Receptor**: این الگو با استفاده از چندین دریافت کننده رویداد، وظیفه را بین آنها پخش می‌کند. این الگو نیز یک الگوی رفتاری است.
- **الگوی State Table**: این الگو با ساخت یک جدول بزرگ از تمامی حالتها، ماشین حالت را می‌سازد. همانطور که در [۴] گفته شده، این الگو از دسته الگوهای آفرینشی است.
- **الگوی State**: این الگو همان الگوی State است که در [۳] آمده است. این الگو یک الگوی رفتاری است.
- **الگوی Decomposed And State**: این الگو برای پیاده سازی حالت های AND طراحی شده است. این الگو یک الگوی رفتاری است.

۵.۲.۳ Safety with Redundancy Patterns

- **الگوی One's Complement**: این الگو با هدف امنیت، داده‌ها را یک بار دیگر به صورت قرینه ذخیره می‌کند. این الگو با ایجاد تکرار باعث امنیت بیشتر می‌شود و یک الگوی رفتاری است.
- **الگوی CRC**: این الگو با اضافه کردن اضافات مربوط به تابع CRC، امنیت را بهبود می‌بخشد. این الگو از نوع رفتاری است.
- **الگوی Channel**: این الگو در بین الگوهای معماری نیز بیان شد. اما ساختار ریزدانه این الگو می‌تواند به عنوان یک الگوی طراحی مطرح شود. این الگو مشابه Chain of Responsibility [۳] است و از نوع الگوهای رفتاری است. این الگو با اضافه کردن تکرار در کانال‌ها می‌تواند به امنیت کمک کند.
- **الگوی Protected Single Channel**: این الگو با اضافه کردن چند چک روی یک کانال، وضعیت امنیت را بهبود می‌بخشد. این الگو در [۴] به عنوان یک الگوی معماری نیز بیان شده اما مسئله این است که اگر به ساختار آن معماری دقت کنیم، می‌بینیم که در اصل آن همان معماری **الگوی Channel** است که الگوی طراحی Protected Single Channel روی آن اعمال شده است و نباید آن را به عنوان یک الگوی معماری مجزا دید.
- **الگوی Dual Channel**: این الگو نیز بر پایه **الگوی Channel** ساخته شده و تفاوت آن در استفاده از دو کانال است که یعنی تکرار را افزایش داده است.

۶.۲.۳ Safety without Redundancy Patterns

- **الگوی Smart Data**: این الگو با ایجاد شرطهایی روی توابع دریافت یک داده، امنیت دسترسی خواندن و نوشتن آن را ارتقا می‌دهد. این الگو از نوع الگوهای رفتاری است.
- **الگوی Smart Pointer**: این الگو با ساخت Smart Pointerها، از مشکلات زیادی که ممکن است با پوینترها اتفاق بیافتد جلوگیری می‌کند. این الگو به شکل یک Wrapper حول اشاره‌گر قرار می‌گیرد و یک الگوی ساختاری است. با توجه به توضیحات داده‌شده، این الگو هم از دسته الگوهای امنیت است و هم از دسته الگوهای مدیریت حافظه.

۷.۲.۳ Memory Patterns

- **الگوی Smart Pointer**: این الگو با ساخت Smart Pointerها، از مشکلات زیادی که ممکن است با پوینترها اتفاق بیافتد جلوگیری می‌کند. این الگو به شکل یک Wrapper حول اشاره‌گر قرار می‌گیرد و یک الگوی ساختاری است. با توجه به توضیحات داده‌شده، این الگو هم از دسته الگوهای امنیت است و هم از دسته الگوهای مدیریت حافظه.
- **الگوی Static Allocation**: این الگو یکی از الگوهای مدیریت حافظه است و نحوه تخصیص منابع را تنظیم می‌کند. این الگو از دسته الگوهای آفرینشی است و این حافظه‌ها را تخصیص می‌دهد.
- **الگوی Pool Allocation**: این الگو نیز هدفی مشابه **الگوی Static Allocation** دارد و یک الگوی آفرینشی است.
- **الگوی Fixed Sized Buffer**: این الگو نیز هدفی مشابه **الگوی Static Allocation** دارد و یک الگوی آفرینشی است.
- **الگوی Garbage Collection**: این الگو با اضافه کردن کلاس GarbageCollector فرایند آزادسازی حافظه را مدیریت می‌کند و یک الگوی رفتاری است.
- **الگوی Garbage Compactor**: این الگو هدفی مشابه **الگوی Garbage Collection** دارد و مانند آن یک الگوی رفتاری است.

۴ مراجع

- Douglass, Bruce Powel. Design patterns for embedded systems in C: an embedded software [۱]
engineering toolkit. Elsevier, 2010.
- Zalewski, Janusz. "Real-time software architectures and design patterns: Fundamental con- [۲]
cepts and their consequences." Annual Reviews in Control 25 (2001): 133-146.
- Gamma, Erich, et al. "Design patterns: Abstraction and reuse of object-oriented design." [۳]
ECOOP'93—Object-Oriented Programming: 7th European Conference Kaiserslautern, Ger-
many, July 26–30, 1993 Proceedings 7. Springer Berlin Heidelberg, 1993.
- Douglass, Bruce Powel. Real-time design patterns: robust scalable architecture for real-time [۴]
systems. Addison-Wesley Professional, 2003.
- Armoush, Ashraf. Design patterns for safety-critical embedded systems. Diss. RWTH Aachen [۵]
University, 2010.
- Buschmann, Frank, et al. Pattern-oriented software architecture: a system of patterns. Vol- [۶]
ume 1. Wiley, 2001.