دانشگاه صنعتی شریف

دانشکده مهندسی کامپیوتر

عنوان:

الگوها در سیستم های نهفته بی درنگ

نویسنده

علی محسنی نژاد

استاد

دکتر رامان رامسین

مرداد ۱۴۰۳

فهرست مطالب

۱ مقدمه

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

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

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

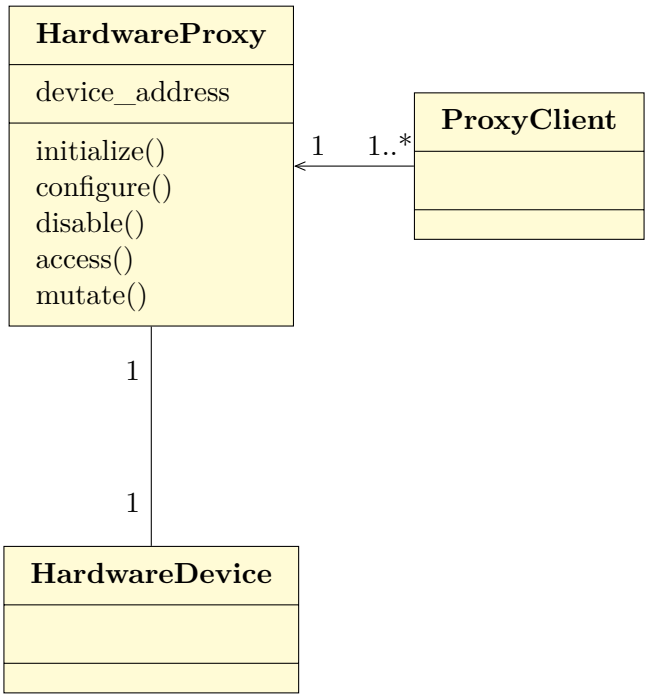
۲ پیشینه پژوهش

۱.۲ الگوهای طراحی برای دسترسی به سخت‌افزار

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

۱.۱.۲ الگوی **Hardware Proxy**

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



شکل ۱: دیاگرام کلاس Hardware Proxy

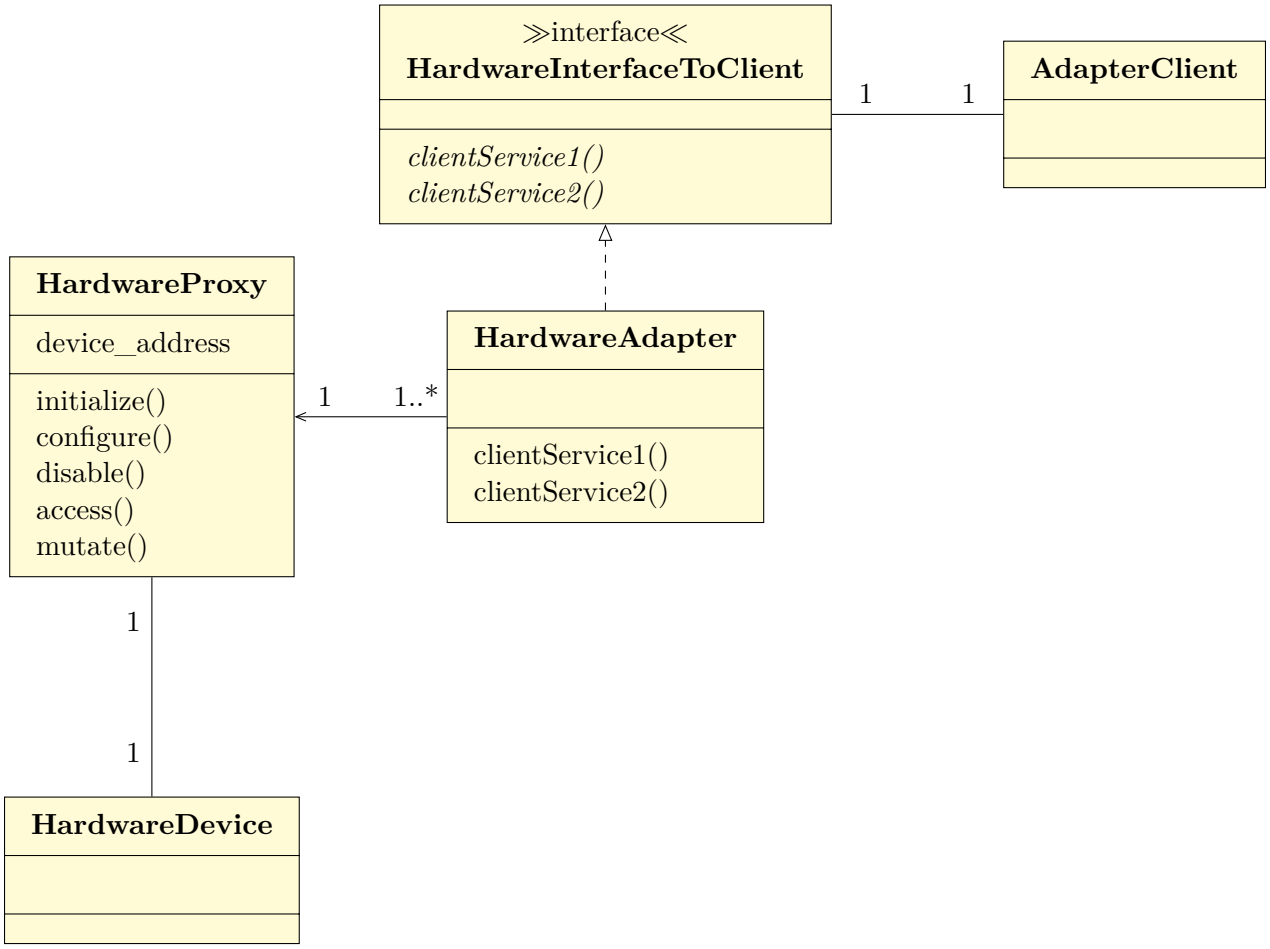
همانطور که در شکل ۱ دیده می‌شود، کلاس پروکسی توابع مشخصی را در اختیار کلاینت‌ها قرار می‌دهد[۱]. توضیحات مربوط به هر یک از توابع کلاس پروکسی در شکل زیر داده شده‌است:

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

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

۲.۱.۲ الگوی **Hardware Adapter**

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



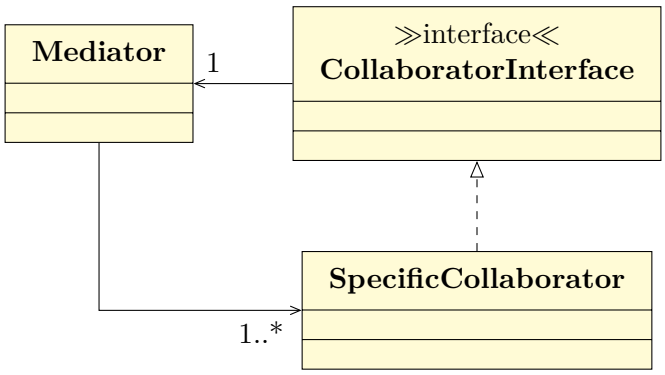
شکل ۲: دیاگرام کلاس Hardware Adapter

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

۳.۱.۲ الگوی **Mediator**

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

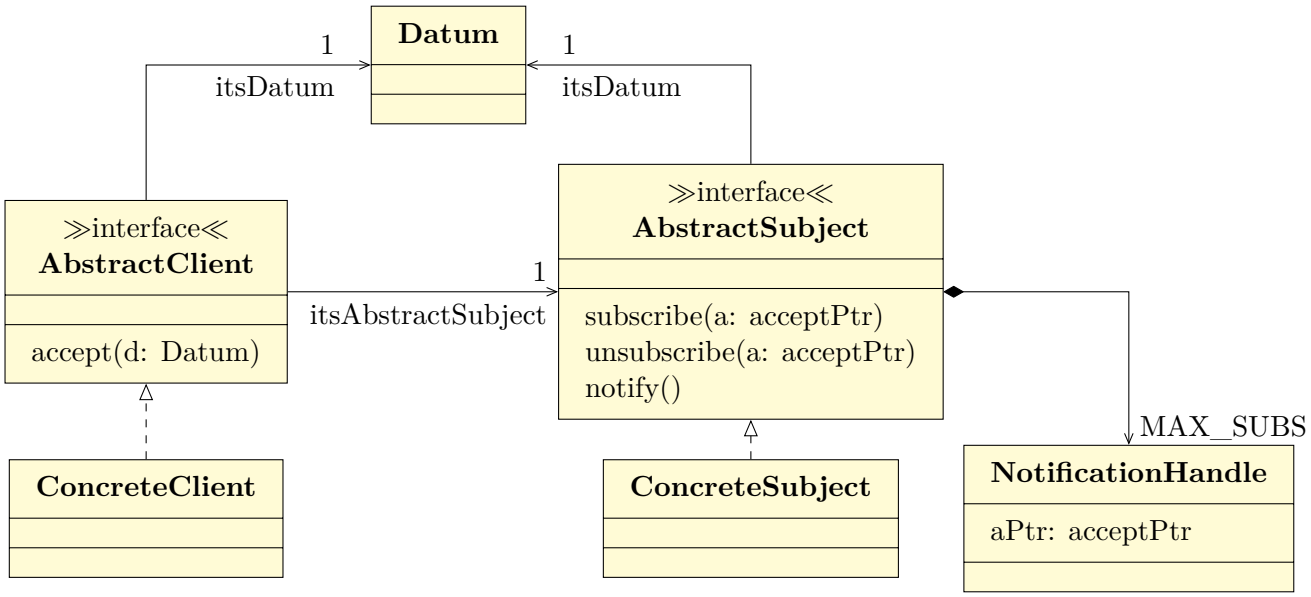
همانطور که در شکل مشخص است، کلاس میانجی با هر یک از کلاس‌های همکار ارتباط دارد. این ارتباط به این شکل است که کلاس میانجی تمامی پیاده‌سازی‌های رابط همکار را می‌شناسد و با آن‌ها ارتباط دارد. این کلاس‌ها خودشان نیز همانطور که نشان‌داده شده، میانجی را می‌شناسند و با آن ارتباط دارند. هر یک از کلاس‌های همکار،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



شکل ۳: دیاگرام کلاس Mediator

۴.۱.۲ الگوی **Observer**

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



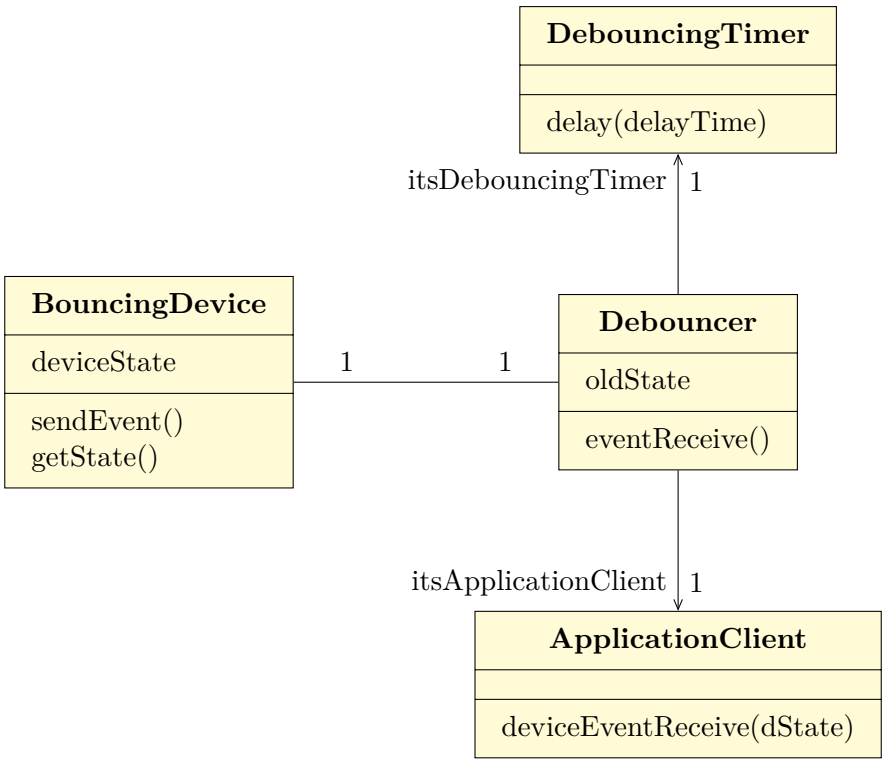
شکل ۴: دیاگرام کلاس Observer

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

۵.۱.۲ الگوی **Debouncing**

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

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



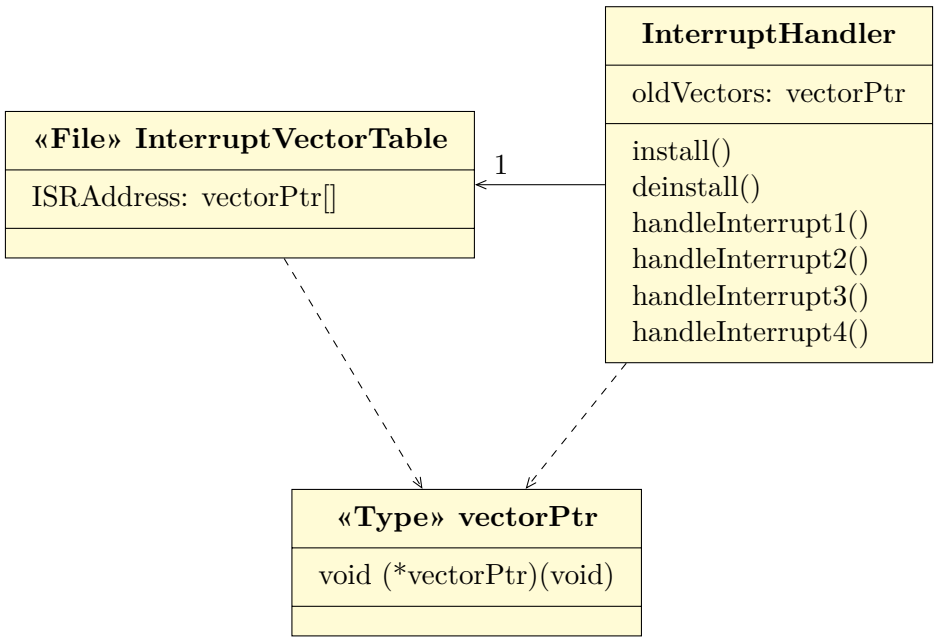
شکل ۵: دیاگرام کلاس Debouncing

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

۶.۱.۲ الگوی **Interrupt**

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

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



شکل ۶: دیاگرام کلاس Interrupt

۷.۱.۲ الگوی Polling

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

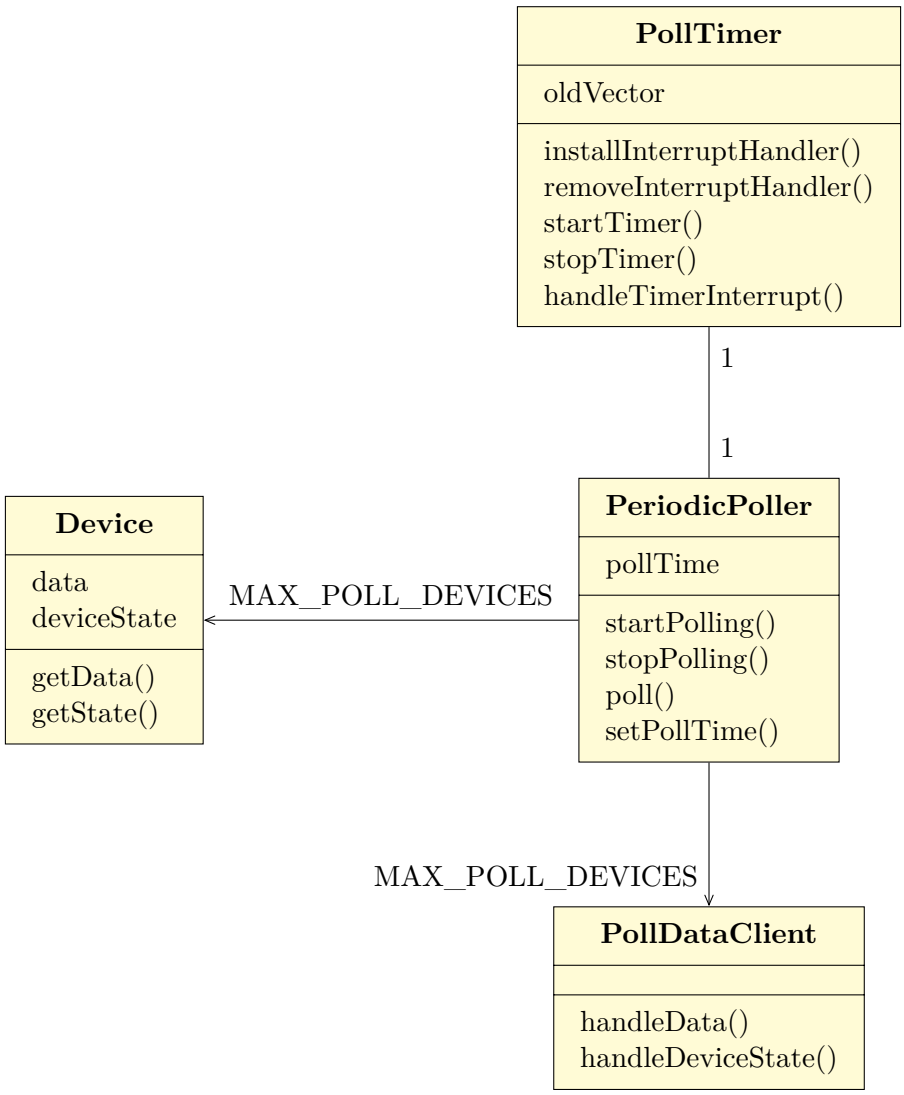
در این الگو کلاس Device همان سخت‌افزار/حافظه/... هست که می‌خواهیم داده‌هایش را دریافت کنیم. کلاس PollDataClient نیز کلاینتی است که می‌خواهد داده‌های Device را دریافت کند. در ساختار شکل ۷، کلاس PeriodicPoller با سرکشی از Device داده‌های آن را دریافت می‌کند. این سرکشی زمانی انجام می‌شود که کلاس PollTimer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

۲.۲ الگوهای طراحی برای همزمانی نهفته و مدیریت حافظه

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

۱.۲.۲ الگوی Cyclic Executive

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



شکل ۷: دیاگرام کلاس Periodic Polling

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

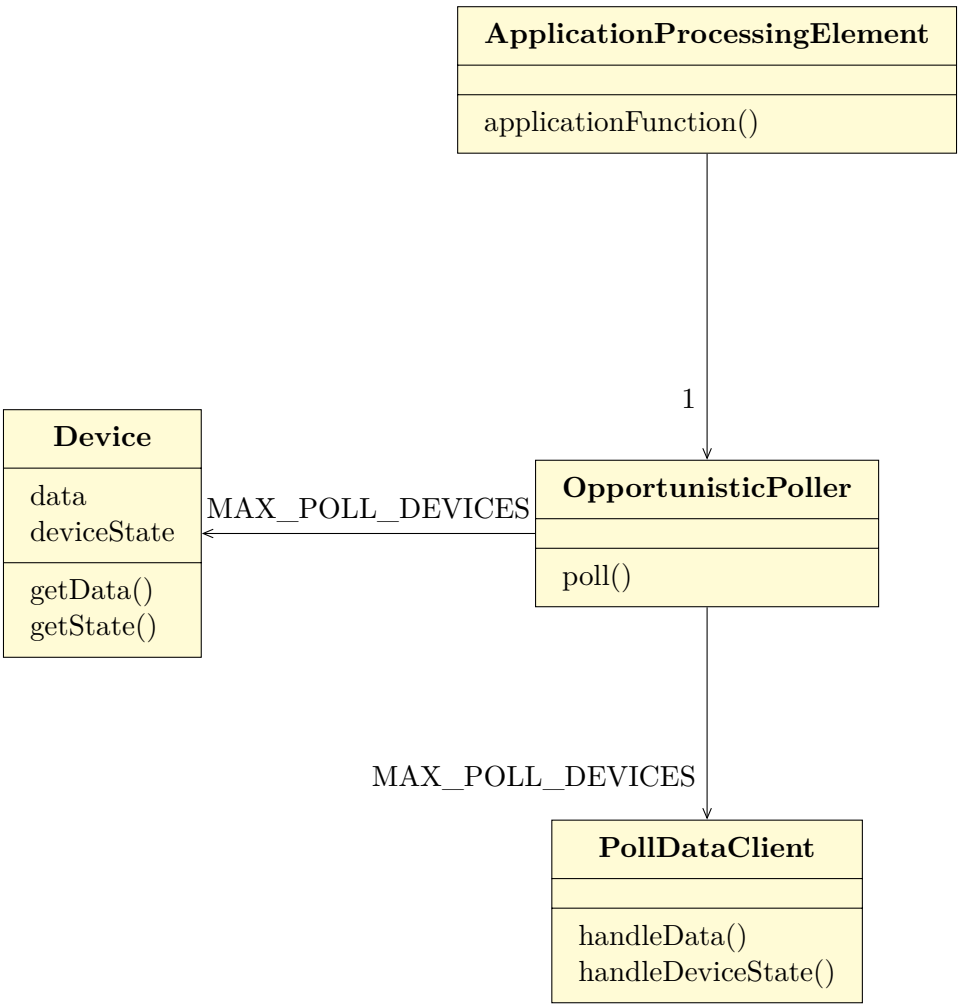
۲.۲.۲ الگوی Static Priority

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

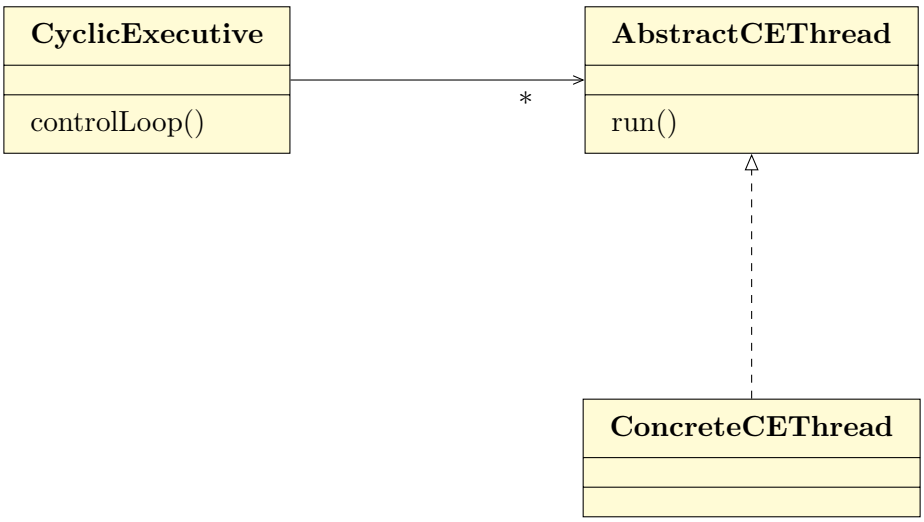
۳.۲.۲ الگوی Critical Region

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

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



شکل ۸: دیاگرام کلاس Opportunistic Polling



شکل ۹: دیاگرام کلاس Cyclic Executive

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

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

۴.۲.۲ الگوی **Guarded Call**

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

۵.۲.۲ الگوی **Queuing**

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

۶.۲.۲ الگوی **Rendezvous**

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

۷.۲.۲ الگوی **Simultaneous Locking**

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

۸.۲.۲ الگوی **Ordered Locking**

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

۳.۲ الگوهای طراحی برای ماشین‌های حالت

۱.۳.۲ الگوی **Single Event Receptor**

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

۲.۳.۲ الگوی **Multiple Event Receptor**

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

۳.۳.۲ الگوی **State Table**

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

۴.۳.۲ الگوی **State**

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

۳ تحلیل

۴ مراجع

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

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

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

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

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