



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

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

۱۵	الگوهای نرم افزاری برای سیستم‌های Safety-Critical	۱۲.۲
۱۵	الگوی N-Version Programming	۱.۱۲.۲
۱۵	الگوی Recovery Block	۲.۱۲.۲
۱۵	الگوی Acceptance Voting	۳.۱۲.۲
۱۵	الگوی N-Self Checking Programming	۴.۱۲.۲
۱۵	الگوی Recovery Block with Backup Voting	۵.۱۲.۲
۱۵	الگوهای ترکیبی سخت افزار و نرم افزار برای سیستم‌های Safety-Critical	۱۳.۲
۱۵	الگوی Protected Single Channel	۱.۱۳.۲
۱۵	الگوی 3-Level Safety Monitoring	۲.۱۳.۲
۱۶	تحلیل	۳
۱۷	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.

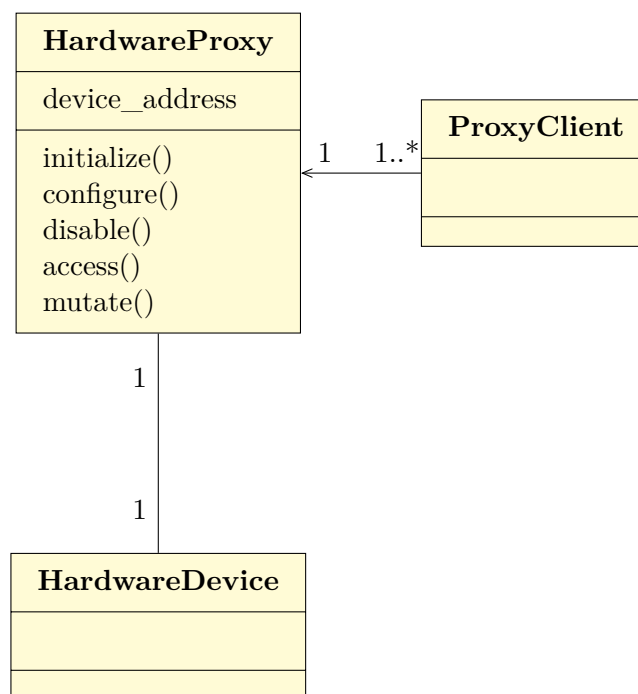
## ۲ پیشینه پژوهش

### 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 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 کلاینتان 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

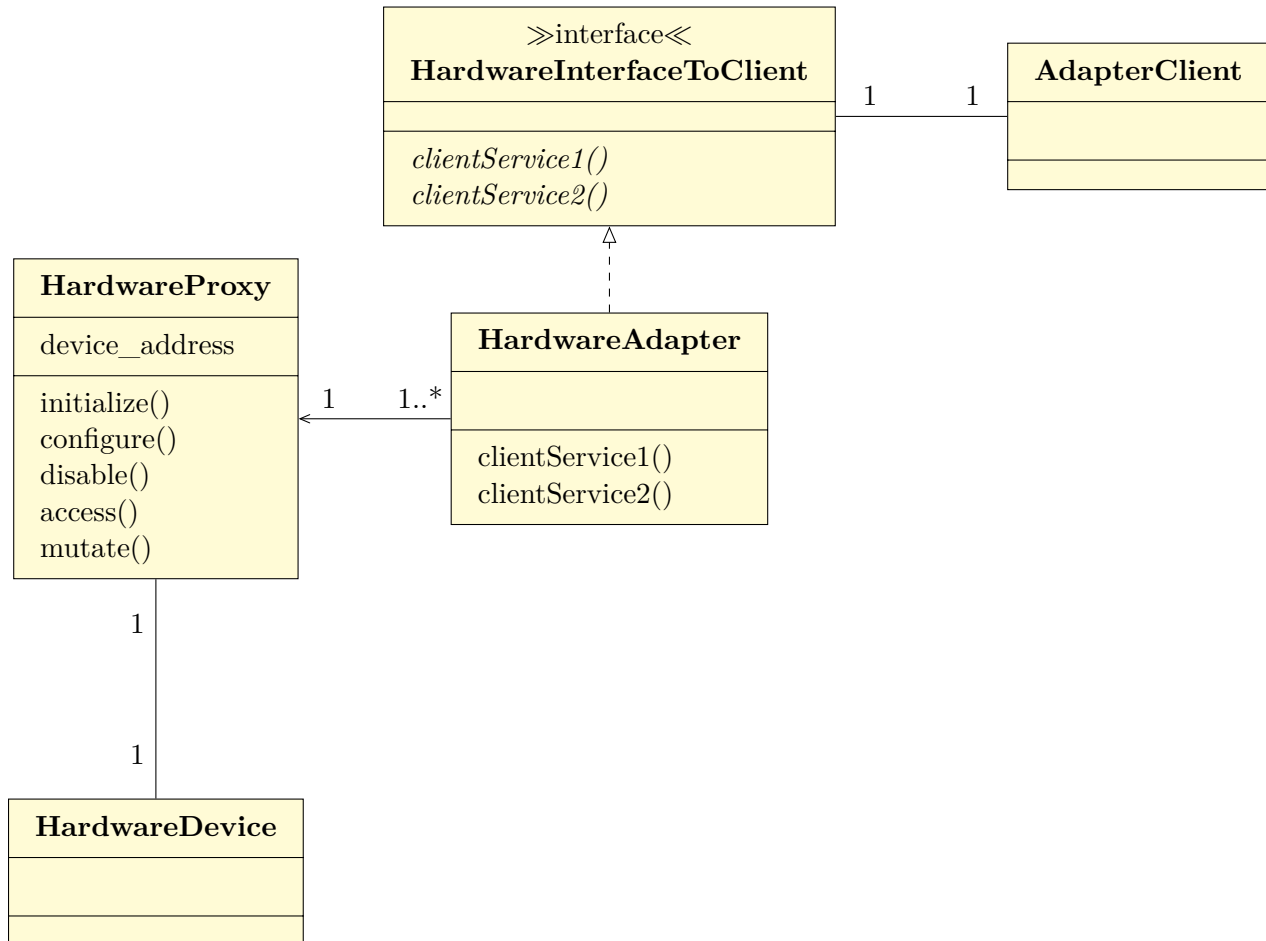
همانطور که در شکل ۱ دیده می شود، کلاس پروکسی توابع مشخصی را در اختیار کلاینت ها قرار می دهد<sup>۱</sup>. توضیحات مربوط به هر یک از توابع کلاس پروکسی در شکل زیر داده شده است:

- 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.

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

### 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 این که متوجه تفاوت‌های آن‌ها شود ارتباط بگیرد. این الگو روی ساختار الگوی Hardware Proxy بنا شده‌است و دیاگرام کلاس آن در شکل ۲ ترسیم شده‌است.



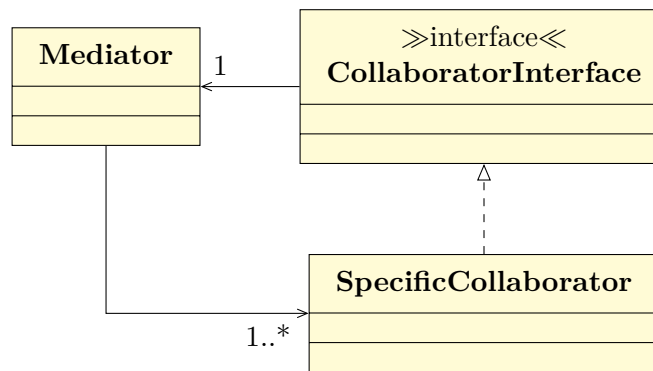
شکل ۲: دیاگرام کلاس Hardware Adapter

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 متناظراً 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.

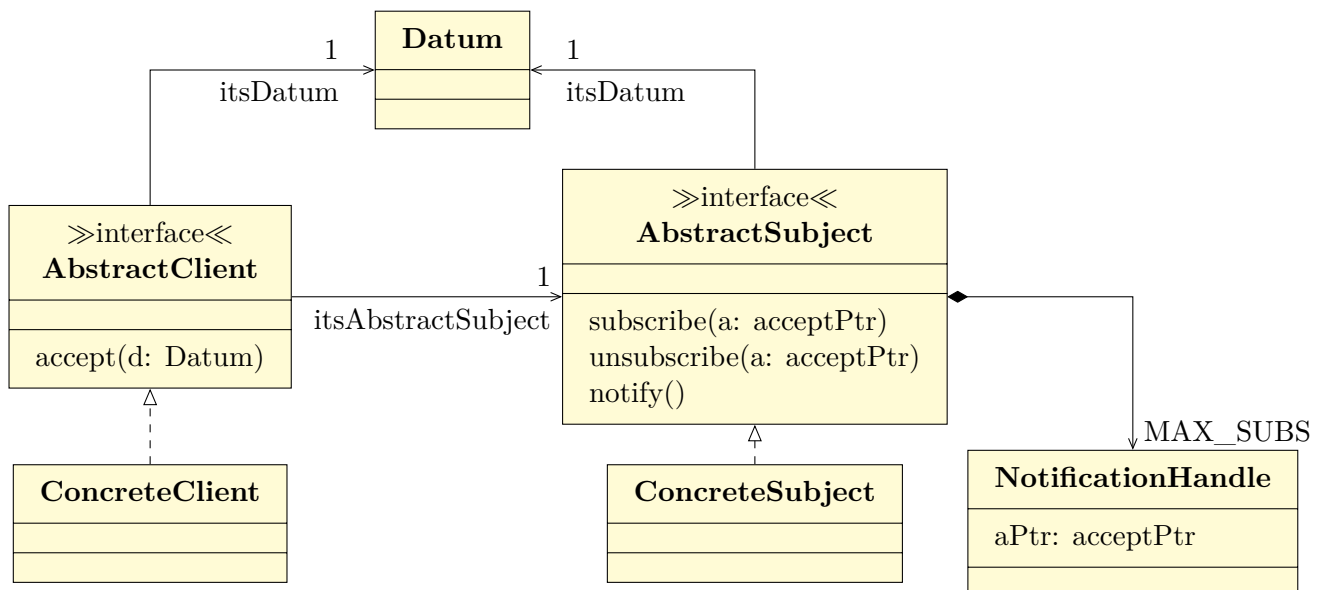
همانطور که در شکل مشخص است، کلاس میانجی با هر یک از کلاس‌های همکار ارتباط دارد. این ارتباط به این شکل است که کلاس میانجی تمامی پیاده‌سازی‌های رابط همکار را می‌شناسد و با آن‌ها ارتباط دارد. این کلاس‌ها خودشان نیز همانطور که نشان‌داده شده، میانجی را می‌شناسند و با آن ارتباط دارند. هر یک از کلاس‌های همکار، 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: دیگرام 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 عضو شده 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 دیگرام کلاس این الگو را می‌بینیم.



شکل ۴: دیگرام کلاس Observer

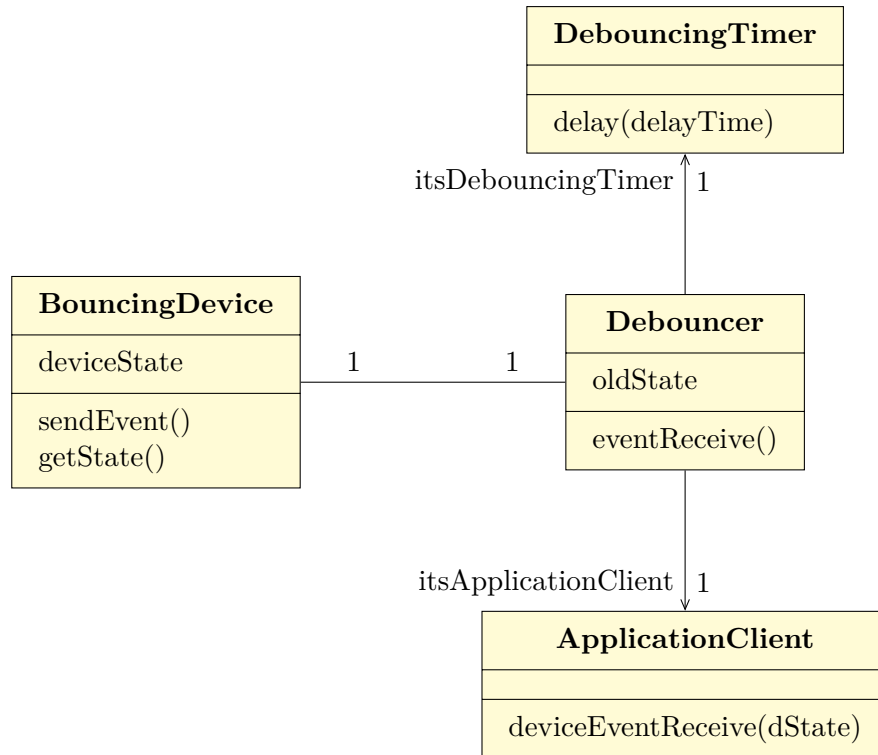
در این ساختار کلاینت‌ها با فرستادن یک اشاره‌گر به کلاس سابجکت، درخواست عضویت برای سرویس می‌فرستند. کلاس سابجکت نیز با ذخیره کردن اشاره‌گرهای مختلف از طرف کلاینت‌ها زمانی که داده جدید آماده می‌شود، با فراخوانی تابع notify، تابع accept که اشاره‌گرهایش را در NotificationHandle قرار داده، با پاس دادن ورودی 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 معمولاً از نوع پروکسی هستند (الگوی [Hardware Proxy](#)).



## ۵.۱.۲ الگوی Debouncing

در سخت افزار بسیاری از ورودی ها به صورت دکمه ها و سویچ هایی هستند که بر اثر ایجاد اتصال دو فلز با یکدیگر، باعث فعال شدن یک پایه شده و آغازگر یک عملیات در نرم افزار 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 می شود. این قطع و وصل شدن، باعث می شود که نتوانیم حالت فعلی سخت افزار را به درستی در نرم افزار ضبط کنیم.

این الگو به ما کمک می کند که با صبر کردن برای یک مدت کوتاه، مقدار ورودی را زمانی که پایدار شده است بخوانیم. با این کار دغدغه معتبر بودن مقدار خوانده شده را در کلاینت نخواهیم داشت. دیاگرام کلاسی این الگو در شکل ۵ نمایش داده شده است.



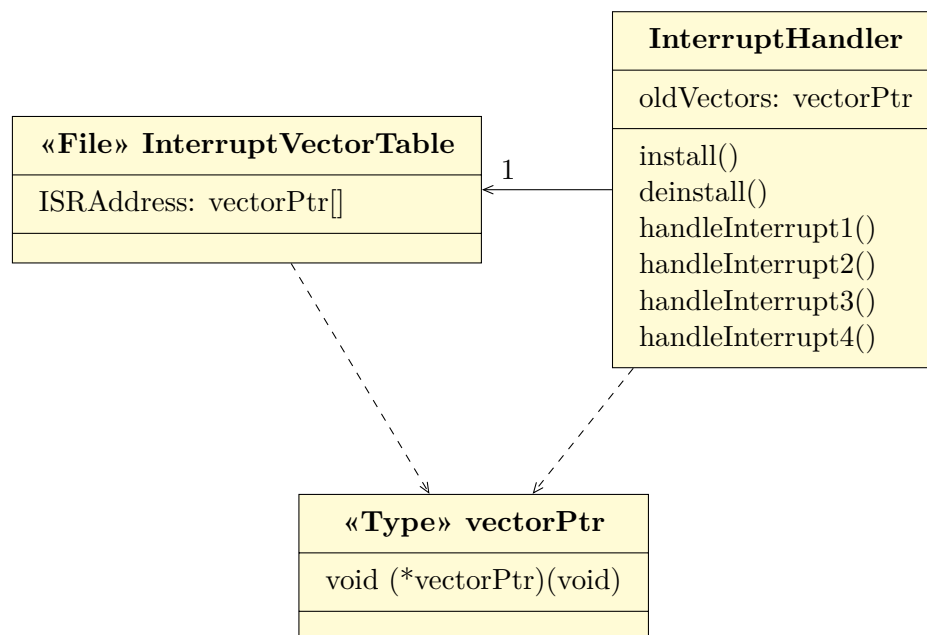
شکل ۵: دیاگرام کلاس Debouncing

در این ساختار، BouncingDevice همان سخت افزار مورد بررسی است. تابع sendEvent می تواند یک نوع اینترپت سخت افزاری باشد که نرم افزار را از تغییر در سخت افزار باخبر می سازد و getState می تواند یک عملیات خواندن از حافظه باشد. کلاس Debouncer وظیفه ارائه حالت پایدار سخت افزار به کلاینت را دارد. این کار با استفاده از یک کلاس زمان سنج انجام می شود که با ایجاد یک تاخیر نرم افزاری تا پایدار شدن شرایط خروجی سخت افزار، خواندن حالت سخت افزار را به تعویق می اندازد.

## ۶.۱.۲ الگوی Interrupt

یکی از واحدهای مهم در سیستم های سخت افزاری، واحد Interrupt است. Interrupt برای هندل کردن وقایعی است که توسط سخت افزار جرقه زده می شوند. زمانی که یک interrupt رخ می دهد، نرم افزار فرایند کاری اصلی خود را متوقف کرده و یک فرایند رسیدگی به interrupt اتفاق افتاده آغاز می شود. با انجام این فرایند و رسیدگی به interrupt، فرایند اصلی نرم افزار دوباره از سر گرفته می شود. ساختار این الگو در شکل ۶ نمایش داده شده است.

در این الگو، کلاس InterruptHandler کار اصلی را انجام می دهد. این کلاس دارای بردار Interruptهاست. با فراخوانی تابع install می توان این بردار را با یک بردار جدید جایگزین کرد. این بردار در اصل تعدادی اشاره گر به توابعی است که در صورت بروز Interrupt باید فراخوانی شوند. با تابع deinstall نیز می توان بردار را به حالت قبلی برگرداند. فایل InterruptVectorTable شامل یک لیست از اشاره گرها به توابع Interrupt Service Routine است. و vectorPtr صرفاً یک نوع اشاره گر به تابعی است که از نوع آن در InterruptVectorTable استفاده شده است.



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 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.

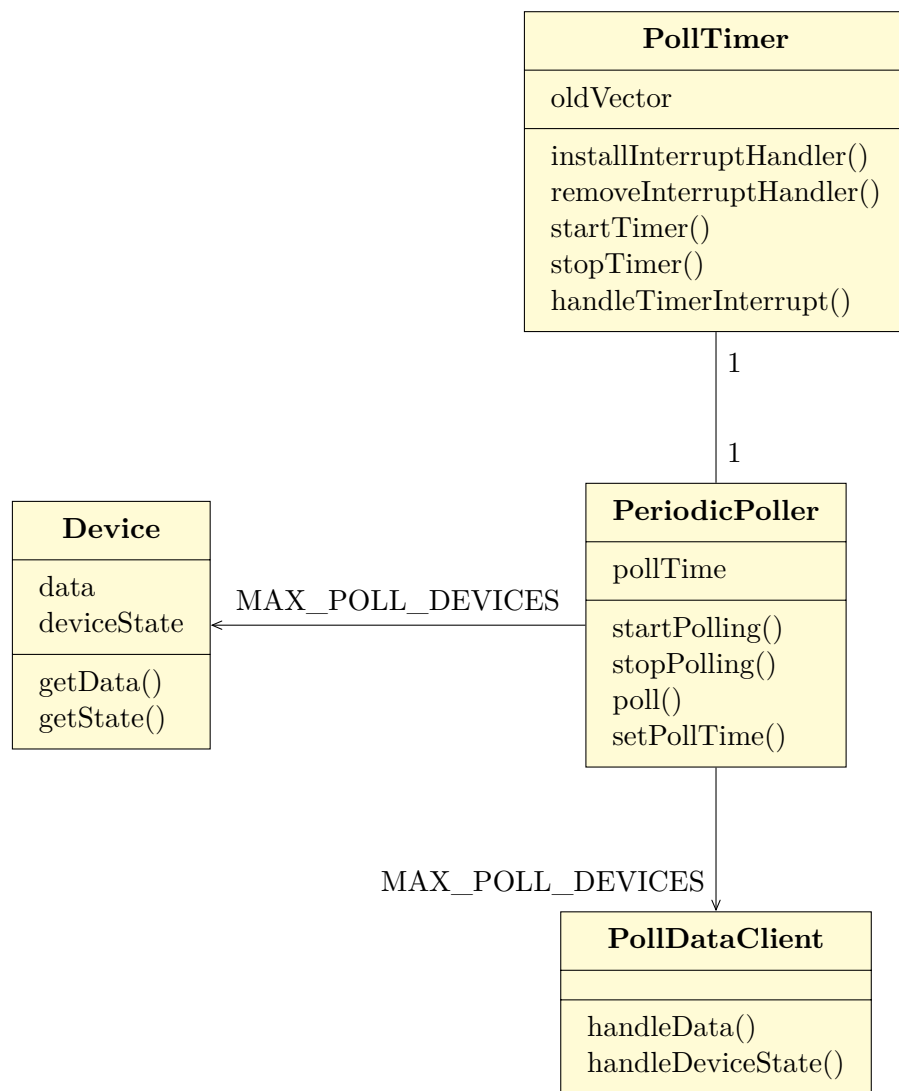
در این الگو کلاس Device همان سخت‌افزار/حافظه... هست که می‌خواهیم داده‌هایش را دریافت کنیم. کلاس PollDataClient نیز کلاینتی است که می‌خواهد داده‌های Device را دریافت کند. در ساختار شکل ۷، کلاس PeriodicPoller با سرکشی از Device داده‌های آن را دریافت می‌کند. این سرکشی زمانی انجام می‌شود که کلاس PollTimer 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 گرفته شود، 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 **AbstractCEThread**هایی که وجود دارد صدا می‌زند.<sup>۲</sup>

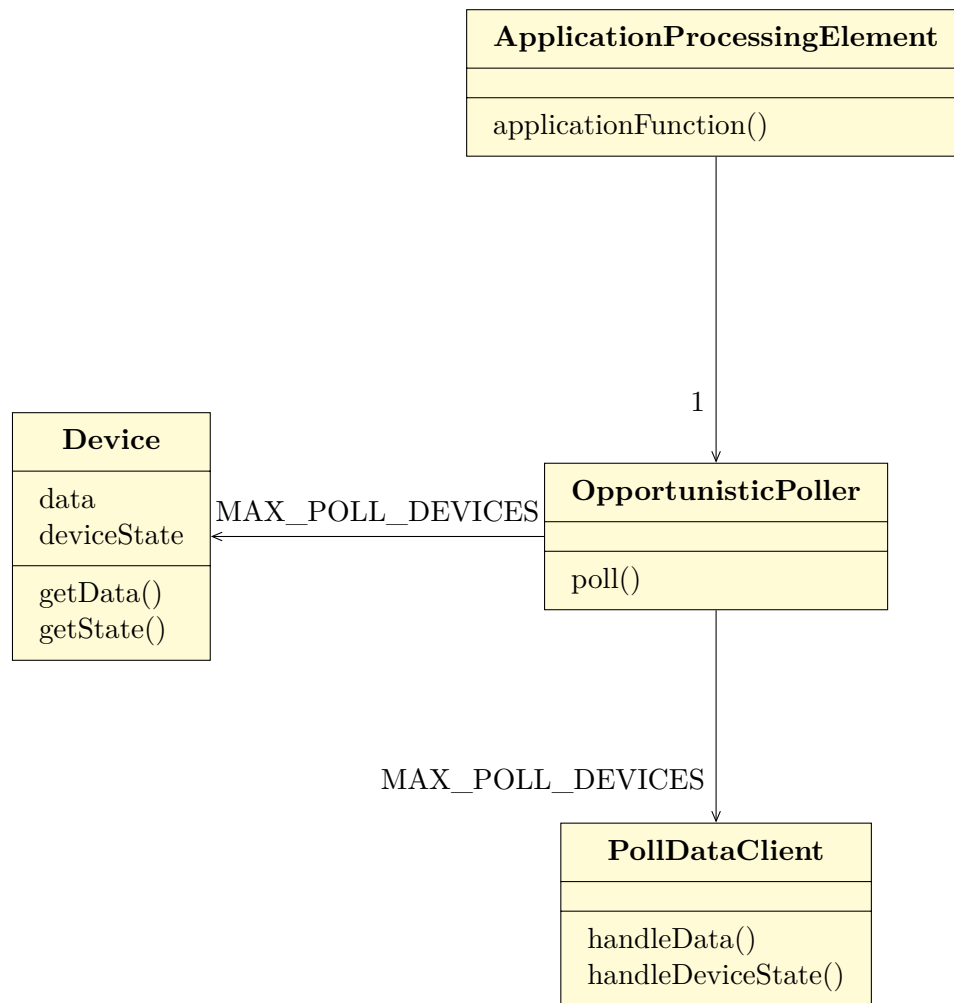
### ۲.۲.۲ الگوی Static Priority

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.

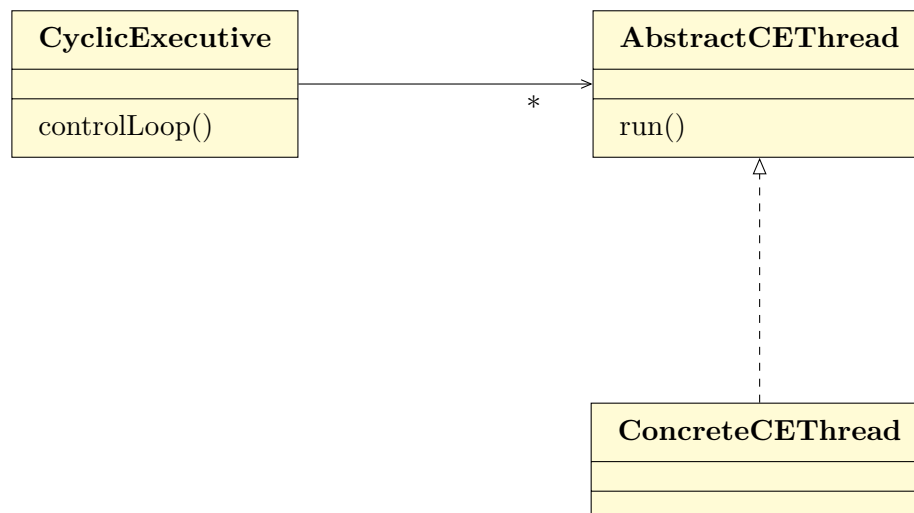
### ۳.۲.۲ الگوی Critical Region

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 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

<sup>۲</sup> در [۸]، 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 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.

#### 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 اشتراک گذاری 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.

#### ۶.۲.۲ الگوی Rendezvous

این الگو 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 سنکرون‌سازی 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.

#### ۷.۲.۲ الگوی Simultaneous Locking

این الگو با هدف جلوگیری از Deadlock به وجود آمده است. این کار را به این روش انجام می‌دهد که اجازه نمی‌دهد هیچ تسکی یک منبع را زمانی قفل کند که خود منتظر آزاد شدن 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 صبر کردن چرخه‌ای تسک‌ها برای یکدیگر را می‌گیرد.



الگوهای طراحی برای ماشین‌های حالت	۳.۲
الگوی Single Event Receptor	۱.۳.۲
الگوی Multiple Event Receptor	۲.۳.۲
الگوی State Table	۳.۳.۲
الگوی State	۴.۳.۲
And States	۵.۳.۲
الگوی Decomposed And State	۶.۳.۲
الگوهای امنیت و قابلیت اطمینان	۴.۲
الگوی One's Complement	۱.۴.۲
الگوی CRC	۲.۴.۲
الگوی Smart Data	۳.۴.۲
الگوی Channel	۴.۴.۲
الگوی Protected Single Channel	۵.۴.۲
الگوی Dual Channel	۶.۴.۲
الگوهای معماری زیربخش‌ها و اجزا	۵.۲
الگوی Layered	۱.۵.۲
الگوی Five Layer	۲.۵.۲
الگوی Microkernel	۳.۵.۲
الگوی Channel	۴.۵.۲
الگوی Recursive Containment	۵.۵.۲
الگوی Hierarchical Control	۶.۵.۲
الگوی Virtual Machine	۷.۵.۲
معماری Component-Based	۸.۵.۲
الگوی ROOM	۹.۵.۲
الگوهای معماری هم‌زمانی	۶.۲
الگوی Message Queuing	۱.۶.۲
الگوی Interrupt	۲.۶.۲
الگوی Guarded Call	۳.۶.۲
الگوی Rendezvous	۴.۶.۲
الگوی Cyclic Execution	۵.۶.۲
الگوی Round Robin	۶.۶.۲
الگوی Static Priority	۷.۶.۲
الگوی Dynamic Priority	۸.۶.۲
الگوهای معماری حافظه	۷.۲
الگوی Static Allocation	۱.۷.۲
الگوی Pool Allocation	۲.۷.۲

## ۳ تحلیل



## 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.