

A look at Software Architecture in C++

With Explanation of Design Patterns and Architectural Patterns

```
class Strategy {  
public:  
    virtual void execute() = 0;  
};  
  
class ConcreteStrategyA :  
    public Strategy {  
    void execute() override {  
        / ... */  
    }  
};
```



SIMPLE EXPLANATIONS
Clear and easy to understand



PRACTICAL EXAMPLES
Real-world, hands-on examples



40+ ESSENTIAL PATTERNS
Proven solutions for modern C++



**INCLUDING ALL
23 GANG OF FOUR (GoF)
DESIGN PATTERNS**



**PLUS ADDITIONAL
ARCHITECTURAL AND
MODERN SOFTWARE PATTERNS**

{ **LEARN BETTER DESIGNS. BUILD BETTER SOFTWARE.** }

Compiled/Edited/Reviewed/Published by Hadi Abbasi (Version 1.0.0)

نگاهی بر معماری نرم افزار

در C++

با تشریح **Design Pattern** ها (الگوهای طراحی)
و **Architectural Pattern** ها (الگوهای معماری)

به زبانی ساده و قابل فهم با مثال های کاربردی
شامل بیش از 40 الگوی مختلف کاربردی
از جمله 23 الگوی کتاب Gang Of Four و برخی موارد دیگر

گردآوری/تحریر/ویرایش/نشر توسط هادی عباسی (نسخه 1.0.0)

بسم الله الرحمن الرحيم



گردآوری/تحریر/ویرایش / نشر توسط هادی عباسی

سایت: <https://hiens.org>

ایمیل: hadi.abbasi.programmer@gmail.com

لینکدین: <https://linkedin.com/in/hadi-abbasi-programmer>

گیت هاب: <https://github.com/HadiAbbasi>

انتشار در 1 مهر 1405 / September 23rd, 2026

فہرست

5	سختن نویسنده در ابتدای کتاب	
7	صفحه	مقدمه
9	صفحه	فصل اول: دیزاین پترن های ساختنی Creational
9	صفحه	Factory Method الگوی
14	صفحه	Abstract Factory الگوی
19	صفحه	Singleton الگوی
23	صفحه	prototype الگوی
31	صفحه	Builder الگوی
38	صفحه	Lazy Initialization الگوی
42	صفحه	Object Pool الگوی
48	صفحه	Multitone الگوی
56	صفحه	Dependency Injection الگوی
72	صفحه	فصل دوم: دیزاین پترن های ساختاری Structural
72	صفحه	Adapter الگوی
82	صفحه	Bridge الگوی
88	صفحه	Composite الگوی
97	صفحه	Decorator الگوی
100	صفحه	Facade الگوی
104	صفحه	Flyweight الگوی
113	صفحه	Proxy الگوی
124	صفحه	Pimple الگوی
129	صفحه	فصل سوم: دیزاین پترن های رفتاری Behavioral
129	صفحه	Chain Of Responsibility الگوی
133	صفحه	Command الگوی
138	صفحه	Interpreter الگوی
163	صفحه	Mediator الگوی
168	صفحه	Memento الگوی (و گریزی بر الگوریتم Time Reversing در بازی Prince Of Persia)
199	صفحه	Observer الگوی (و گریزی بر الگوی Publish/Subscriber در جهت مدیریت Event ها)
204	صفحه	State الگوی
208	صفحه	Strategy الگوی
214	صفحه	Template Method الگوی
221	صفحه	Visitor الگوی
230	صفحه	Iterator الگوی
235	صفحه	Null Object الگوی
242	صفحه	فصل چهارم: الگوهای سازمانی (Enterprise Patterns)
243	صفحه	Repository الگوی
251	صفحه	Service Locator الگوی
265	صفحه	Unit Of Work الگوی
273	صفحه	فصل پنجم: الگوهای معماری (Architectural Patterns)
273	صفحه	MVC معماری
281	صفحه	MVP معماری
286	صفحه	MVVM معماری
289	صفحه	Domain Driven Design (DDD) معماری
324	صفحه	Clean, Onion, Hexagonal (Port & Adapter) معماری های
375	صفحه	CQRS معماری
388	صفحه	ECS مختص بازی ها و موتورهای بازی (Naive ECS, Archetype ECS, Sparse Set ECS) معماری
429	صفحه	سرفصل های موضوعات مختلف:
436	صفحه	بخش پایانی نویسنده:

سخن نویسنده در ابتدای کتاب به نام خدای بزرگ و یکتا، پدید آورنده هستی بخش

با سلام و احترام

در فاصله سال‌های ۱۳۸۸ تا ۱۳۹۱ افتخار داشتم به عنوان مدیر انجمن فلش و AS3 وبسایت «مجید آنلاین» با نام‌های کاربری مختلف همچون عناوین HadiMetal و ++Hadi++ فعالیت کنم. حضور در آن جامعه تخصصی و آشنایی با جمعی از برنامه‌نویسان و توسعه‌دهندگان حرفه‌ای، تجربه‌ای ارزشمند و الهام‌بخش برای من بود.

پس از کسب این تجربه موفق، با الگوبرداری از ساختار وبسایت PCSeven و با همراهی تعدادی از دوستان حرفه‌ایم در سایت مجید آنلاین، ایده راه‌اندازی یک جامعه تخصصی خصوصی را مطرح کردم. حاصل این ایده، وبسایت FLXpert.com بود؛ سایتی با ساختار انجمن فروم مبنای آن برای اعضای منتخب و خبره انجمن فلش مجید آنلاین در دسترس قرار داشت.

عضویت در این مجموعه صرفاً از طریق دعوت‌نامه امکان‌پذیر بود و تنها افرادی که به واسطه دانش فنی، فعالیت‌های ارزشمند و کسب رتبه‌های بالا در انجمن مجید آنلاین شناخته شده بودند، امکان حضور در آن را پیدا می‌کردند.

هدف اصلی این پروژه، ایجاد بستری برای تولید دانش تخصصی بود؛ به گونه‌ای که هر یک از اعضا در حوزه تخصصی خود به تحقیق، تولید محتوا و انتشار مطالب بپردازد تا سایر اعضا نیز از آن بهره‌مند شوند. همچنین ساختار سایت به گونه‌ای طراحی شده بود که کاربران، متناسب با سطح دانش و رتبه خود، به لایه‌های عمیق‌تر و پیشرفته‌تر محتوا دسترسی پیدا کنند و به مرور زمان مسیر رشد علمی خود را طی نمایند.

با وجود استقبال اولیه، این پروژه به دلیل نبود انگیزه‌های مالی و عدم استمرار مشارکت اعضا، به موفقیت بلندمدت دست پیدا نکرد. در آن دوران، شخصاً فعالیت گسترده‌ای در زمینه‌های مختلف از جمله Unity3D, Unreal Engine 3 (UDK), Qt Framework و HTML5 داشتم و آموزش‌های متنی و ویدیویی متعددی تولید و منتشر می‌کردم. اما پس از گذشت چند ماه، عملاً در کنار معدودی از دوستان، به تنها اعضای فعال مجموعه تبدیل شدیم و همین موضوع باعث شد انگیزه ادامه مسیر را از دست بدهم و فعالیت سایت متوقف شود.

با این حال، یکی از ارزشمندترین دستاوردهای آن گروه، ترجمه گروهی کتابی چندصد صفحه‌ای و بسیار تخصصی درباره Design Pattern ها در زبان ActionScript 3 (زبانی با ساختاری نزدیک به Java) بود. در این پروژه، حدود ۲۰ تا ۳۰ نفر از اعضا هر کدام مسئول ترجمه ۵ تا ۱۰ صفحه از کتاب شدند و در نتیجه، کل کتاب در مدت مناسبی ترجمه گردید. البته به دلیل تفاوت سبک نگارش و ادبیات مترجمان، ویرایش نهایی کتاب زمان قابل توجهی را به خود اختصاص داد و در نهایت هم به سرانجام نرسید.

در آن دوران، برخلاف هدف اولیه گروه که اشتراک‌گذاری دانش میان اعضای حرفه‌ای بود، پروژه به درخواست کل اعضا، سمت انتشار و فروش کتاب سوق پیدا کرد. این تجربه برای من نکته مهمی را روشن ساخت؛ اینکه در بسیاری از پروژه‌های اجتماعی و انجمن‌های تخصصی، وجود انگیزه‌های مالی می‌تواند نقش بسیار مهمی در تداوم فعالیت و موفقیت بلندمدت اعضا ایفا کند.

اکنون نزدیک به پانزده سال از آن دوران می‌گذرد. در این سال‌ها، همواره احساس می‌کردم باید یک بار کل این الگوها را یاد بگیرم و مطالب مختلفی از تمام این الگوها جمع‌آوری کردم! ولی در حین مطالعه و منظم کردن مطالب جمع‌آوری شده، احساس کردم جای یک مرجع جامع و کاربردی برای گردآوری الگوهای طراحی و معماری و مفاهیم مرتبط در زبان C++ خالی است. همین موضوع انگیزه‌ای شد تا تجربه‌ها، مطالعات و منابع مختلف را یک‌بار دیگر با دقت مرور کرده و در قالب کتابی جامع گردآوری کنم. البته در نگارش این کتاب از آن کتاب ترجمه شده استفاده‌ای نکردم چرا که فایلش را هم گم کرده‌ام! از دیگر سو، کتاب پیش روی شما، غیر از الگوهای طراحی، به الگوهای معماری و سازمانی و ECS هم می‌پردازد!

این کتاب حاصل بیش از دو سال و نیم (به صورت ناپیوسته)، مطالعه، تحقیق، گردآوری، دسته‌بندی، ویرایش و نگارش مستمر است و امیدوارم بتواند منبعی مفید برای علاقه‌مندان و توسعه‌دهندگان نرم‌افزار باشد و دریچه‌ای ارزشمند برای یادگیری مطالب بیشتر در این زمینه‌ها برایتان بگشاید!

اهداف انتشار این کتاب

این کتاب به صورت رایگان منتشر شده است و اهداف اصلی من از نگارش و انتشار آن عبارت‌اند از:

1. کسب رضایت پروردگار متعال از طریق پرداخت زکات دانش و کمک به رشد علمی برنامه‌نویسان و توسعه‌دهندگان.
2. تعمیق و تثبیت آموخته‌های شخصی از طریق تحقیق، نگارش و آموزش.
3. به اشتراک گذاشتن تجربیات و دانش با جامعه برنامه‌نویسان.
4. تقویت رزومه و سوابق حرفه‌ای خود.

پیش‌نیازهای مطالعه کتاب: برای بهره‌گیری بهتر از مطالب این کتاب، آشنایی با موارد زیر توصیه می‌شود:

- تسلط بر زبان برنامه‌نویسی C++11 یا نسخه‌های جدیدتر +تمپلیت ها (Generic Programming)
- آشنایی با مفاهیم برنامه‌نویسی شیء‌گرا (OOP)
- شناخت اصول SOLID (می‌توانید به کتابچه نوشته شده اینجانب با موضوع Solid در وبلاگم در hiens.org مراجعه فرمایید!)
- آشنایی با Smart Pointer ها و Raw Pointer ها.

• آشنایی مختصر با معماری نرم افزار و UML و Microservice ها و Domain Driven Design و Clean Architecture و Onion Architecture و ...

• آشنایی با برخی از ساختمان داده ها و کانتینرهای استاندارد C++ مانند: std::map و std::list و std::unordered_map و ...

نکته بسیار مهم: در فرآیند تحقیق، گردآوری، بازبینی و تکمیل مطالب این کتاب، از دستیارهای هوش مصنوعی از جمله: **DeepSeek**، **Qwen**، **ChatGPT** و **Claude** نیز به عنوان ابزارهای کمکی استفاده شده است و بخش های قابل توجهی از متون این کتاب عینا مطالب دریافت شده از این ربات ها می باشد! همچنین تصویر جلد این کتاب را هم با یاری جستن از **ChatGPT** و **Qwen** طراحی کردم!

همچنین از دوست عزیز و قدیمی ام **آقای محمد آزاد**، استاد عزیز درس معماری نرم افزارم با رویکرد مایکروسرویس ها در آموزشگاه تحلیل داده تشکر می کنم که استخوان بندی بخش قابل توجهی از مطالب این کتاب را از کلاس ارزشمند ایشان وام گرفته ام! البته من قطره ای از دریای مطالب ارزشمند ایشان را در این کتاب گنجانده ام! پیشنهاد می کنم برای یادگیری جزئیات معماری نرم افزار در مایکروسرویس ها در کلاس ایشان (آموزشگاه تحلیل داده) شرکت کنید! ضمناً در ابتدا قصد ما نوشتن کتاب معرفی و تشریح **Design Pattern** های تیم 4 نفره **Gof** بود ولی بعداً تصمیم گرفتیم به سمت الگوهای سازمانی و معماری هم پیش برویم! لذا در ابتدا **Design Pattern** ها (شامل 23 الگوی **Gof** و موارد دیگر) را تشریح کردیم و سپس برخی از معماری های مهم سازمانی و معماری (در مجموع کل الگوها، بیش از 40 مورد) را نیز تشریح می کنیم! همچنین در ابتدا، قصد بنده از نوشتن مطالب این کتاب، داشتن رفرنسی شخصی برای انواع الگوهای طراحی بود ولی نهایتاً در حین ویرایش، به فکر انتشار آن در قالب یک کتاب آموزشی افتادم! لذا تلاش کردم لحن و نگارش آنرا با حفظ ادبیات نوشتاری، از حالت رسمی خارج کنم و تا حدودی ساده و بی تکلف پیش ببرم و به جای صفحه آرای حرفه ای، صفحات ساده **Office Word** را در قالب **PDF** خروجی بگیرم و منتشر نمایم! اگر در ابتدای کتاب، بیش از توضیح معماری نرم افزار، به توضیح الگوهای طراحی می پردازم، به جهت نیت اولیه کتاب است که به خواست پروردگار یکتا، در نسخه بعدی اصلاح خواهد شد!

برخی از دوستان پیشنهاد ارائه آموزش های ویدیویی داده بودند! اما ذکر 2 نکته حائز اهمیت است:

1- اینک در سطح وب هزاران هزار آموزش ویدیویی در همه زمینه ها وجود دارد.

2- نهایتاً پس از دیدن آموزش های مختلف ویدیویی، مجبورید مطالب یادگرفته را جایی یادداشت کنید! این **PDF** می تواند همان مطالب یادداشت شده شما باشد! کما اینکه خود من هم یادداشت هایم را به این کتاب تبدیل کردم!

گروهی از دوستان به تغییر سبک برنامه نویسی با **Vibe Coding** و نقش **Agent** های هوش مصنوعی اشاره می کردند و معتقد بودند دیگر نیازی به یادگیری الگوهای برنامه نویسی به سبک گذشته نیست. به نظر من، اتفاقاً برنامه نویس یا معمار نرم افزاری که می خواهد بهترین خروجی را از **Agent** ها بگیرد، بهتر است خودش با این الگوها آشنا باشد؛ حتی مطالعه یک کتاب 400 صفحه ای در این زمینه می تواند کیفیت **Prompt** ها و در نهایت خروجی **Agent** را به شکل محسوسی ارتقا دهد.

ضمن اینکه حتی سازندگان همین **Agent** ها نیز در نهایت به کد تولیدشده توسط **AI** بسنده نمی کنند؛ تغییرات و کدهای تحول آفرین در خود این سیستم ها توسط مهندسان انسانی بررسی، ارزیابی و تأیید می شوند. بنابراین، به بیان ساده، حتی کدهایی که **ChatGPT** را متحول می کنند نیز در نهایت مهر تأیید انسان را پشت سر گذاشته اند!

شاید دوستان با دیدن ایموجی کنار تیترا، مرا به استفاده از هوش مصنوعی محکوم کنند در صورتیکه امروزه استفاده از ابزارهای هوشمند، جزو لاینفک تمام پروژه های جهان است! کما اینکه در ویدیوهای آموزش برنامه نویسی 10 سال پیش، گروهی از اساتید به جهت استفاده بیش از اندازه از **Copy/Paste** مدام از مخاطبان عذرخواهی می کردند، شاید به این جهت که آن زمان، بسیاری معتقد بودند برنامه نویس فقط باید با سرعت بالا تایپ کند و اینک در عصر مدرن دوران این گونه عقاید جاهلیت تمام شده است و مهم، کیفیت کد و خروجی کد است! لذا من با افتخار در بسیاری از بخش های کتاب از دوره های آموزش ویدیویی و هوش مصنوعی و ... استفاده کرده ام و البته بخش هایی از آن حاصل تجربیات خود بنده است!

از اینکه این کتاب را برای مطالعه انتخاب کرده اید، صمیمانه سپاسگزارم. شاید این کتاب در آینده کامل تر شود! در پایان این کتاب، بخش هایی که ان شاء الله در آینده تکمیل خواهد شد، نوشته شده است!

امیدوارم مطالب ارائه شده بتواند سهمی هر چند کوچک در رشد علمی شما به سوی موفقیت بیشتر در مسیر توسعه نرم افزار داشته باشد.

با آرزوی موفقیت و سربلندی همه شما، هادی عباسی

انتشار در 1 مهر 1405 / September 23rd, 2026

در دهه ۹۰ با گسترش برنامه‌نویسی شیء‌گرا، توسعه‌دهندگان نرم‌افزار در پروژه‌های گوناگون با مسائل طراحی مشابهی مواجه شدند که هر یک راه‌حل‌های مجزایی برای آن‌ها ارائه می‌دادند. به مرور، این چالش‌ها به موضوعی رایج در جوامع مهندسی نرم‌افزار تبدیل شد و راه‌حل‌های آن‌ها مورد ارزیابی و تبادل نظر قرار گرفت.

در سال ۱۹۹۴، چهار نویسنده که بعدها به گنگ آو فور (GoF) مشهور شدند، با انتشار کتاب *Design Patterns: Elements of Reusable Object-Oriented Software*، ۲۳ مورد از این چالش‌های متداول را به همراه راه‌حل‌های استاندارد و اثبات‌شده‌ی آن‌ها گردآوری کردند. این اثر، پایه‌گذار مفهومی شد که امروزه به عنوان الگوهای طراحی (Design Patterns) در ادبیات مهندسی نرم‌افزار شناخته می‌شود! در واقع این الگوها همچون جعبه ابزاری برای مسایل مشترکی است که خواه ناخواه در نرم‌افزارهای بسیاری مورد استفاده قرار می‌گیرند!

اصطلاحات اولیه:

آنتی پترن:

شما برای مشکلی از روش یا دیزاین پترنی استفاده می‌کنید که شاید از دیدگاه اولیه مشکل شما را حل کند ولی مشکلات بدتری در فازهای دولوپ پروژه پدید می‌آورد. اصطلاحاً مثل چکش طلا که چکش است منتها حیفتان می‌آید با آن به چیزی بکوبید چون از طلاست!

چارچوب استاندارد تحلیل و مستندسازی دیزاین پترن‌ها

برای تحلیل دقیق هر الگوی طراحی، باید آن را از ابعاد مختلف کالبدشکافی کرد. موارد زیر، پارامترهای اصلی در آنالیز یک دیزاین پترن هستند:

1- شناسنامه و دسته‌بندی (Classification & Scope)

دسته اصلی: (Category) الگو در کدام گروه بنیادین قرار می‌گیرد؟

سازنده: (Creational) تمرکز بر مکانیزم‌های ایجاد اشیا.

ساختاری: (Structural) تمرکز بر ترکیب کلاس‌ها و اشیا برای تشکیل ساختارهای بزرگتر.

رفتاری: (Behavioral) تمرکز بر ارتباطات و تخصیص مسئولیت بین اشیا.

همروندی: (Concurrency) تمرکز بر الگوهای چند thread-ی و مدیریت حالت‌های هم‌زمان.

سطح کاربرد: (Scope/Level) الگو در چه مقیاسی پیاده‌سازی می‌شود؟

سطح کلاس: (Class-level) تمرکز بر روابط استاتیک و ارث‌بری مانند الگوهای GoF

سطح شیء: (Object-level) تمرکز بر روابط دینامیک و Composition

سطح کامپوننت/معماری: (Component/Architecture-level) تمرکز بر ماژول‌ها و ساختار سیستم مانند الگوهای POSA مثل MVC یا Layers

2- تحلیل بستر و مسئله (Context & Problem Analysis)

بستر: (Context) وضعیت و شرایطی که الگو در آن کاربرد دارد (مکانی که نیاز به الگو احساس می‌شود).

مسئله: (Problem) چالش یا گره طراحی که در آن بستر رخ می‌دهد و نیاز به راه‌حل دارد.

نیروها: (Forces) نیروها به معنای اشکالات آینده نیستند! نیروها همان قیدها، محدودیت‌ها و اهداف متضادی هستند که باید balance شوند.

مثلاً: "نیروی ۱: نیاز به Decoupling بالا" در مقابل "نیروی ۲: نیاز به Performance بالا". الگوی طراحی تعادلی بین این نیروها ایجاد می‌کند.

3- ارائه راه‌حل (Solution & Intent)

هدف و منظور: (Intent/Purpose) الگو دقیقاً چه کاری انجام می‌دهد؟ چه مزیتی نسبت به راه‌حل‌های جایگزین دارد؟

ساختار راه‌حل: (Solution Structure) توصیف انتزاعی از اجزا (کلاس‌ها/کامپوننت‌ها)، مسئولیت‌ها و نحوه تعامل آن‌ها با یکدیگر برای حل مسئله.

4- تنوع و پیاده‌سازی (Variants & Implementation)

انواع و تغییرپذیرها: (Variants/Known Variations) سبک‌های پیاده‌سازی مختلف یک الگو.

 **مثال:** در الگوی Observer، آیا از سبک عمومی ناشر- مشترک (Pub/Sub) استفاده می‌کنیم یا سبک مصرف‌کننده-تولیدکننده (Producer/Consumer)؟ آیا Push-based است یا Pull-based؟

5- ارزیابی و پیامدها (Evaluation & Consequences)

موارد کاربرد: (Applicability) چه زمانی (کی)، کجا و چگونه باید از این الگو استفاده کنیم؟ (و چه زمانی نباید استفاده کنیم).

- پیامدها، مزایا و معایب: (Consequences, Pros & Cons) نتایج حاصل از اعمال الگو.

 **مزایا:** چه چیزی را بهبود می‌بخشد؟ (مثلاً انعطاف‌پذیری، تست‌پذیری).

 **معایب:** Trade-offs چه هزینه‌ای دارد؟ (مثلاً افزایش مصرف حافظه، پیچیدگی بیشتر در کد، افت جزئی در سرعت).

6- مستندسازی و نمونه‌ها (Documentation & Examples)

کاربردهای شناخته‌شده: (Known Uses) این الگو در کدام کتابخانه‌های استاندارد، فریم‌ورک‌های معروف یا سیستم‌های بزرگ دنیا پیاده‌سازی شده است؟ (مثلاً استفاده از الگوی Composite در ساختار DOM مرورگرها یا فایل سیستم‌ها!)

توضیحات کلی الگوهای طراحی:

دیزاین پترن‌ها یا الگوهای نرم افزار، راه حل‌های کلی و قابل استفاده مجدد برای مشکلات رایجی هستند که در طراحی و توسعه نرم افزار پیش می‌آیند. آنها به عنوان بهترین شیوه‌های شناخته شده برای حل انواع مشکلات شناخته شده اند و راهی برای ارتباط بین توسعه دهندگان درباره راه حل‌های طراحی موثر فراهم می‌کنند.

دیزاین پترن‌ها را می‌توان به سه دسته تقسیم کرد:

1- دیزاین پترنهای ساختنی (Creational):

این دسته از دیزاین پترن‌ها، به فرآیند ایجاد اشیاء متمرکز می‌شوند و آنرا انتزاعی می‌کنند. آنها به ساخت یک سیستم کمک می‌کنند که مستقل از اینکه اشیاء آن چگونه ساخته شده، ترکیب شده و نمایش داده شده اند، مدیریت شوند.

2- دیزاین پترنهای ساختاری (Structural):

این دسته از دیزاین پترن‌ها، به سلسله مراتب (Hierarchy) و نحوه ترکیب کلاس‌ها و اشیاء برای تشکیل ساختارهای بزرگتر متمرکز می‌شوند. دیزاین پترنهای ساختاری-کلاسی، از ارث بری برای ترکیب رابط‌ها یا پیاده‌سازی‌ها استفاده می‌کنند.

3- دیزاین پترنهای رفتاری (Behavioral):

این دسته از دیزاین پترن‌ها، به رفتار و ارتباط بین اشیاء و تخصیص مسئولیت‌ها بین اشیاء توجه می‌کنند. دیزاین پترنهای رفتاری، الگوهایی را توصیف می‌کنند که نه فقط الگوهای اشیاء یا کلاس‌ها، بلکه الگوهای ارتباط بین آنها را نشان می‌دهند. این الگوها، جریان کنترل پیچیده‌ای را مشخص می‌کنند که در زمان اجرا دشوار است تا دنبال شود.

الگوهای سازمانی (Enterprise Patterns):

این دسته از الگوها، مجموعه‌ای از راه‌حل‌ها و الگوهای معماری/طراحی هستند که برای حل مسائل رایج در سیستم‌های سازمانی (Enterprise Applications) به کار می‌روند؛ سیستم‌هایی که معمولاً پیچیده، بزرگ، چندکاربره و دارای نیازهای جدی مثل تراکنش، امنیت، یکپارچگی و مقیاس‌پذیری هستند!

الگوهای معماری (Architectural Patterns):

این دسته از الگوها، بیش از آنکه راهکاری استاندارد و تست شده جهت حل مشکلی خاص باشند، ساختاری استاندارد برای استخوان بندی یک نرم افزار هستند و تفاوت این نوع الگوها با الگوهای طراحی، در نوع استفاده از آنها است! به بیان بهتر دیزاین پترن‌ها یا الگوهای طراحی برای حل مشکل خاصی به کار می‌روند، ولی الگوهای معماری، قالب بندی کلی یک نرم افزار را تعریف می‌کنند!

این کتاب در 5 فصل طراحی شده است. در فصل اول به تشریح الگوهای ساختنی و در فصل دوم به الگوهای ساختاری و در فصل سوم به الگوهای رفتاری می‌پردازیم و در فصل چهارم هم چند مورد از الگوهای سازمانی و در فصل پنجم هم چند مورد از الگوهای معماری مهم را تشریح می‌کنیم. امیدوارم از خواندن کتاب لذت ببرید و مرا دعا بفرمایید...

فصل اول

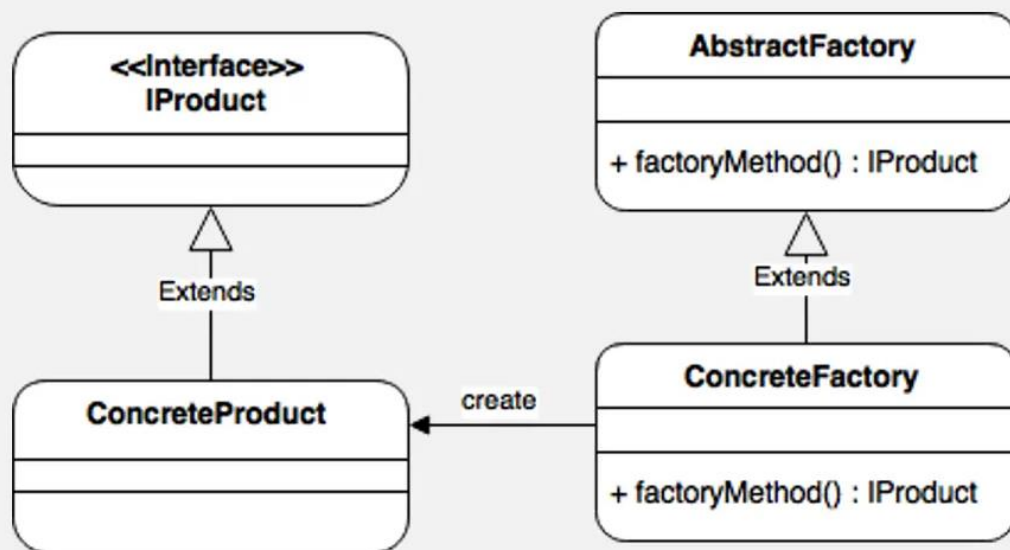
دیزاین پترنهای ساختنی (Creational)

این دسته از دیزاین پترن‌ها، به فرآیند ایجاد اشیاء متمرکز می‌شوند و آنرا انتزاعی می‌کنند. آنها به ساخت یک سیستم کمک می‌کنند که مستقل از اینکه اشیاء آن چگونه ساخته شده، ترکیب شده و نمایش داده شده اند، مدیریت شوند. انواع دیزاین پترنهای ساختاری عبارتند از:

○ **Factory Method**: پترنی از نوع Creational است!



Factory Method – Class diagram



The Factory Method pattern provides a generalized way to create instances of an object and can be a great way to hide implementation details for derived class

اگر در یک بازی یا برنامه، چند گروه از کلاسها را داشته باشیم که هر گروه subclass یک کلاس abstract یا superclass خاص باشد، می توانیم برای هر گروه از مجموعه کلاسها، تابع یا کلاس فکتوری یا کارخانه مخصوص آن نوع را بسازیم. فرض کنید در یک بازی، گروه کاراکترهای انسان و گروه کاراکترهای ربات و گروه ماشین ها و گروه سلاح ها را دارید و هر گروه، از یک کلاس abstract خاص خود ارث بری دارد. می توانید برای این مثال کلاس فکتوری سلاح و کلاس فکتوری کاراکتر انسان و کلاس فکتوری کاراکتر ربات و کلاس فکتوری ماشین بسازید. سپس برای هر فکتوری، یک تابع create می سازید و این تابع، مشخصات مشترک و کامل همه انواع subclass ها را بر اساس کانسٹرکتورها می گیرد و نوعی از کلاس پدر abstract یا غیر abstract را به صورت اشاره گر بیرون می دهد که صد البته با Down Cast یعنی dynamic_cast در C++ قابل تبدیل به فرزند خود است. هدف از این دیزاین پترن، جلوگیری از انتشار ساخت اشیای یک دسته از موجودیت ها در جای جای کد است. چرا که ساخت اشیا را می توانید به یک متد create از factory class مربوطه بدهید تا در صورت نیاز به تغییر، آن را در همین متد تغییر دهید تا تغییرات در جای جای کد نیاز به اصلاح نداشته باشد.

شما می توانید بر اساس کانسٹرکتور مشترک یا تکاملی دسته مربوطه، به متد create پارامتر ورودی دهید و برای هر دسته، enum مخصوص آن دسته را بنویسید تا با نوع enum مشخص کنید کدام کلاس فرزندی را می خواهید بسازید! مثلاً در مثال زیر، ما کلاس abstract shape و کلاسهای فرزندی مثلث و مربع و دایره را داریم.

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
class Shape {
public:
    virtual string getName() = 0;
};

class Triangle : public Shape {
public:
    string getName() { return "Triangle"; }
};

class Square : public Shape {
public:
    string getName() { return "Square"; }
};

class Circle : public Shape {
public:
    string getName() { return "Circle"; }
};
```

حال یک enum به عنوان نماینده هر کدام از انواع shape ها می سازیم:

```
Enum SHAPE_Type{ TRIANGLE, SQUARE, CIRCLE};
```

و کلاس ShapeFactory را به این صورت می نویسیم:

```
// responsible for creating and returning Shape objects on demand...
class ShapeFactory {
public:
    Shape* create(SHAPE_Type t)    // factory method
    {
        switch (t){
            case TRIANGLE:
                return new Triangle();
            case SQUARE:
                return new Square();
            case CIRCLE:
                return new Circle();
        }
    }
};
```

حال هر نمونه ای از فرزندان کلاس shape را می توان با دادن enum نماینده آن نوع، تولید کرد. اگر کانستراکتور shape و subclass هایش نیاز به پارامترهای اضافه داشته باشد، آنها را از طریق متد create مد نظر دریافت می کنیم و به کانستراکتور هر کلاس فرزندی پاس می دهیم.

```
int main()
{
    vector <Shape*> shapes;
    // Our factory is responsible for creating Shape objects on demand
    Factory f;
    // Lets create 3 different shape objects and assign their memory addresses to our shapes vector
    shapes.push_back(f.create(TRIANGLE));
    shapes.push_back(f.create(SQUARE));
    shapes.push_back(f.create(CIRCLE));
    // Iterate over each shape object, calling it's getName method using it's pointer
    for (Shape*& s : shapes)
        cout << s->getName() << endl;
    // cleanup
    for (Shape*& s : shapes)
        delete s;
    shapes.clear();
    return 0;
}
```

 نمونه دیگری از کلاس factory:

```
#include <string>
#include <memory>
class IGameObject {
public:
    virtual ~IGameObject() {}
    virtual void Update() = 0;
    virtual void Render() = 0;
};
class Plane : public IGameObject {
public:
    Plane() {}
    void Update() {}
    void Render() {}
};
class Boat : public IGameObject {
public:
    Boat() {}
    void Update() {}
    void Render() {}
};
```

```
enum class ObjectType { PLANE, BOAT };

std::shared_ptr<IGameObject> MakeGameObjectFactory(ObjectType type) {
    if (type == ObjectType::PLANE) {
        return std::make_shared<Plane>();
    }
    else if (type == ObjectType::BOAT) {
        return std::make_shared<Boat>();
    }
    return nullptr;
}

int main() {
    std::shared_ptr<IGameObject> myObject = MakeGameObjectFactory(ObjectType::PLANE);
    std::shared_ptr<IGameObject> myObject2 = MakeGameObjectFactory(ObjectType::BOAT);
    return 0;
}
```

در این مثال با استفاده از اشاره گر smart از نوع `std::shared_ptr`، داده های ساخته شده از کلاس فرزندی را بی نیاز به `delete` کردن کردیم!

در روشی متفاوت با الگوی Factory Method مطرح شده، `C++` نوین روش جدیدی را برای ساخت انواع اشیای مختلف همراه با پاس دادن انواع داده های مجاز به آن را ارائه کرده است که برای تکمیل این بخش، آنرا نشان می دهیم! البته در این روش شما باید لیست آرگمان های ورودی را متناسب با پارامترهای مجاز Constructor کلاس در حال ساختتان (که به عنوان تمپلیت اول پاس می دهید) ارسال کنید، در غیر اینصورت به خطاهای Compile برخورد خواهید کرد!

🌸 نمونه ای از Factory با Template و Variadic Arguments:

```
#include <memory>
#include <utility>

class Factory {
public:
    template<typename T, typename... Args>
    static std::unique_ptr<T> create(Args&&... args) {
        return std::make_unique<T>(
            std::forward<Args>(args)... // دستور فوروارده، خاصیت ال ولیو و آر ولیو را حفظ می کند
        );
    }
};

// برای مثال اگر بخواهید کلاس پلیر را با ورودی نام هیرو و عدد 100 به کانستراکتورش بسازید
auto player = Factory::create<Player>("Hero", 100);
// در اینجا T برابر Player است و "Hero" و 100 به Constructor آن ارسال می شوند
// شما اینک می توانید به کمک کلاس فکتوری، از هر نوع شی با پاس دادن مقادیر مختلف نمونه سازی کنید
```

نکته مهم 1: روش بهینه در Factory Class، ورودی از نوع رشته json است که همیشه شما را به وابستگی به یک جمله رشته ای json ملزم می کند و شما در رشته json مربوطه می توانید پارامتر های مد نظر خود را تنظیم کنید تا کلاس factory مد نظر با دریافت و parse جیسون شما، پارامتر های مد نظر تان را بازیابی کرده و شی مربوطه را بسازد. این نکته از این بابت حائز اهمیت است که هیچ وقت امضای تابع create تحت تاثیر تغییرات بنیادی در کانستراکتور کلاس shape و فرزندانش تغییر نمی کند و با این کار، تغییرات را از کد مجزا کرده اید!

نکته مهم 2: همیشه سعی کنید یک مقدار default از نوع nullptr در انتهای متد create به اصطلاح return کنید تا اگر به اشتباه، نوع ورودی با هیچ کدام از انواع کلاسهای فرزندی مطابقت نکرد، مقدار nullptr را return کند تا با بررسی not null بودن خروجی، جلوی خطاهای ناخواسته را بگیریم.

نکته 3: می توانید کلاس **factory** خود را به صورت **singleton** در بیاورید تا همه جا با **Call** کردن متد **static** موسوم به **create** از هر نوع فکتوری، شی مورد نظرتان را بسازید.

نکته 4: در **C++** عادت کنید، اشاره گرهای ساخته شده از فرزندان یک کلاس در یک **factory class** را از نوع **smart pointer** بسازید تا در صورت فراموش شدن **delete** کردن اشیا، خود به خود پاک شوند تا در حافظه نمانند و باعث **Memory Leak** نشوند!

نکته 5: از جمله مزایای کلاس **Factory** در آنست که اگر در قانون ایجاد اشیا یک نوع خاص از کلاسها، قانونتان تغییر کند، می توانید آن قانون را درون تابع سازنده اعمال کنید و از نشر آن در همه جای کد جلوگیری کنید! برای مثال اگر قرار باشد به خاطر موضوعی، تمام اشیا ساخته شده از انواع ربات های بازیتهان را در لیست به خصوصی نگه دارید و پس از نابودی هر رباتی، آنرا از لیست مربوطه حذف کنید، در اینجا می توانید اضافه شدن هر ربات در لیست مربوطه را درون تابع **Create** اعمال کنید! وگرنه اگر قرار می شد در چندجای کد، شی از نوع ربات بسازید، باید بلافاصله در خط بعد از ساخت ربات، آنرا به لیست مربوطه اضافه می کردید و این تکرار مکررات اشتباه است!

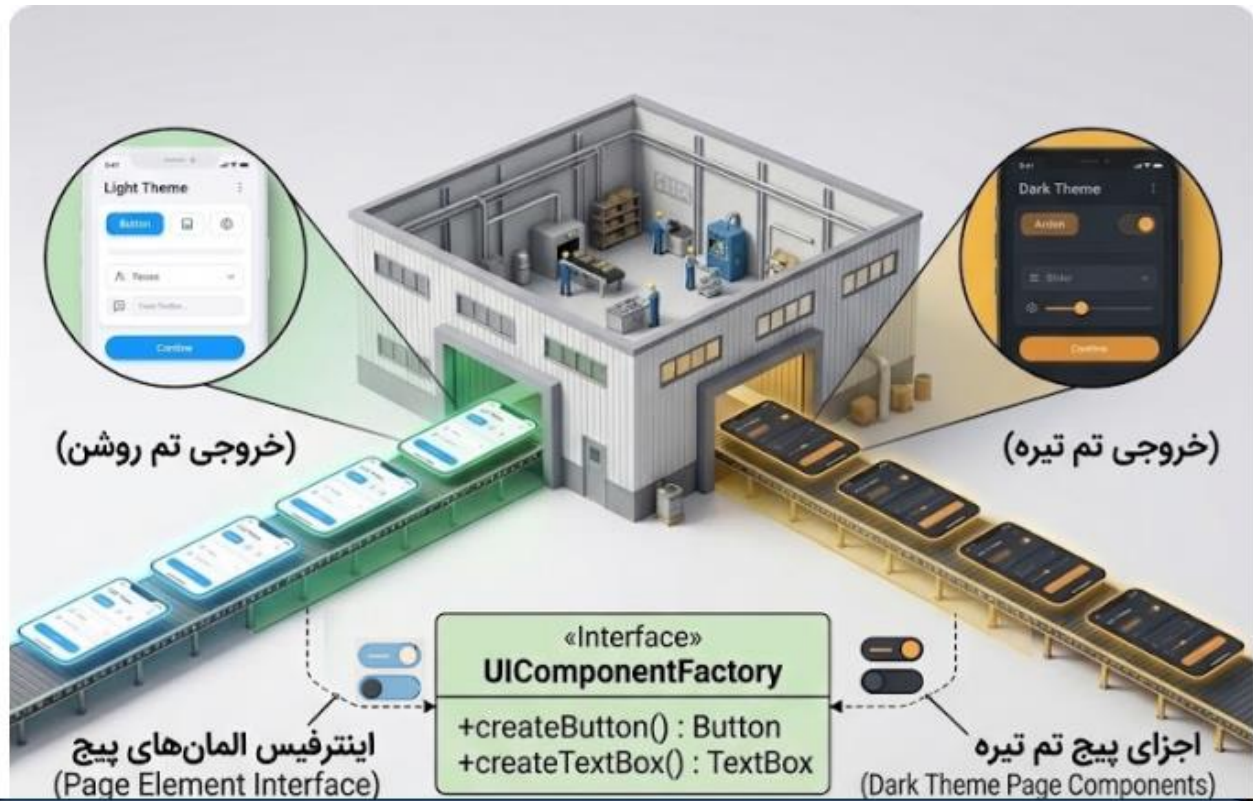
مزایای Factory Pattern: ساخت اشیا در یک جا انجام می شود و متد **create** فکتوری شما، ساخت اشیا را مخفی می کند و با این کار، باعث جلوگیری از انتشار تغییرات در ساخت اشیا در نقاط مختلف کد می شود. از طرفی کار این کلاس و این متد صرفاً یک کار **single responsible** است و آن ساخت اشیا از نوع فرزندان یک کلاس **abstract** است. همچنین فرض کنید چاشنی ساخت یک شی نیاز به ساخت چندین شی دیگر و چندین عملیات گوناگون و نهایتاً تولید شی نهایی داشته باشد و لذا بدون **Factory Class** یا **Factory Function** باید کدهای تکراری با تعداد خطهای زیاد را در جای جای پروژه تکرار کنید و با کوچکترین تغییر این پروسه، فاجعه رخ خواهد داد! چون باید جای جای تکرار کدها را بیابید و تغییرات مورد نیاز را اعمال کنید! در صورتیکه با داشتن یک تابع **Factory**، تغییرات در یک جا اعمال خواهد شد و تمام!

سوال: برای حفظ **Dependency Inversion** اگر کلاس سطح بالای شما مجبور باشد چندین شی از کلاس سطح پایین به تعداد مورد نیازش بسازد، در صورتیکه مجبور باشید کلاس سطح پایین را در قالب یک **Interface** به آن پاس دهید، چگونه می تواند نمونه شی سطح پایین را درون لیست خود **New** کند در صورتیکه به اصل کلاس دسترسی ندارد و فقط به **Interface** یا **Abstract** آن دسترسی دارد؟

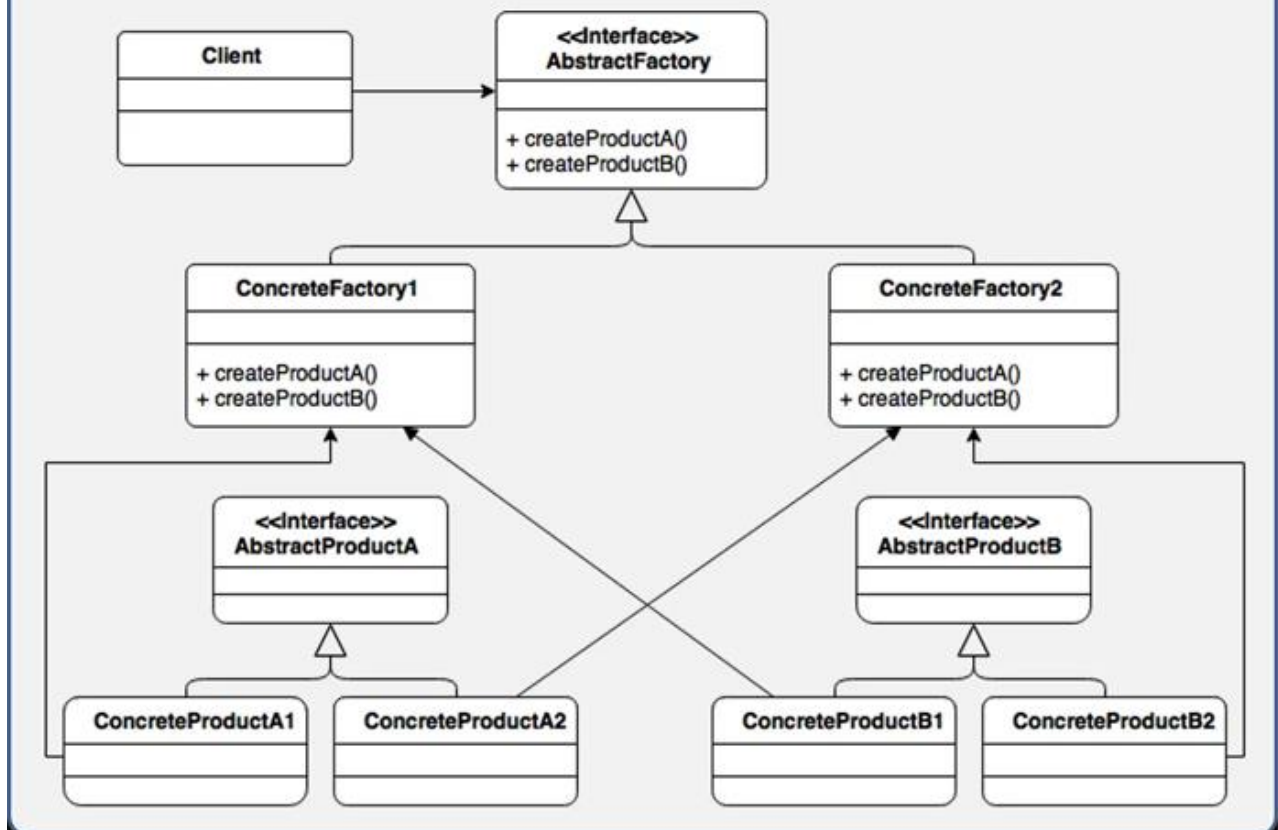
پاسخ: کلاس **Factory** کلاس سطح پایین را به کلاس سطح بالا تزریق کنید! (برای حفظ **Dependency Inversion**، به شکل **Interface** یا **Abstract** از نوع **factory**، به آن تزریق کنید!) سپس از متد **Create** شی **Factory** تزریق شده به تعداد دلخواه استفاده کنید!

معایب Factory Pattern: برای انواع داده هایی که هر کدام بر اساس **abstract** کلاس خود کتگوری بندی می شوند باید یک کلاس فکتوری اضافه کنید. اگر کلاس فکتوریتان با استفاده از **enum** نوع درخواستی شما را تشخیص می دهد هم باید یک **enum** به اضافه یک فکتوری متد یا فکتوری کلاس برای آن اضافه کنید و این یعنی بیشتر کردن مراحل که صد البته به مزایایش می ارزد! ولی درد آنرا در افزودن انواع جدید در ساختار برنامه حس خواهید کرد! آنجایی که باید **Factory class** و **Factory Interface** را هم بسازید!

Abstract Factory: پترنی از نوع Creational است!



Abstract Factory – Class diagram



پترن **Abstract Factory** یکی از قدرتمندترین پترن‌های ساخت (Creational) در طراحی نرم‌افزار است. این پترن به ما اجازه می‌دهد خانواده‌های مرتبط از اشیاء را بدون مشخص کردن کلاس‌های واقعی‌شان ایجاد کنیم. در واقع، **Abstract Factory** یک کارخانه‌ای از کارخانه‌ها است: یک رابط انتزاعی برای ایجاد مجموعه‌ای از اشیاء مرتبط یا وابسته فراهم می‌کند، بدون اینکه نیاز باشد کلاس‌های واقعی را بدانیم.

🌸 چه زمانی از **Abstract Factory** استفاده کنیم؟

از این پترن زمانی استفاده می‌شود که :

- نیاز داریم چندین محصول مرتبط را با یک رابط یکپارچه ایجاد کنیم.
- می‌خواهیم کد را از کلاس‌های ملموس (concrete) جدا کنیم تا قابلیت نگهداری و توسعه بالاتری داشته باشیم.
- نیاز به تغییر راحت بین خانواده‌های مختلف از محصولات داریم.

فرض کنید یک سیستم سفارش قطعات اتومبیل و سر هم بندی آنها را داریم که باید بر اساس برند، در ماشین و موتور را تولید کند. دو برند متفاوت داریم :

- بنز (Mercedes)
- تویوتا (Toyota)

هر کدام از این برندها، اجزای داخلی متفاوتی دارند که باید با هم سازگار باشند. مثلاً نباید موتور بنز را با در تویوتا ترکیب کرد.

در اینجا، هر برند یک خانواده‌ی محصول است و هر بخش (در، موتور) یک محصول در آن خانواده .

🏠 ساختار کلاس‌ها

1. محصولات انتزاعی (Abstract Products)

```
class Door { // درب انتزاعی
public:
    virtual ~Door() = default;
    virtual void describe() const = 0;
    virtual void open() = 0;
};
// موتور انتزاعی
class Engine {
public:
    virtual ~Engine() = default;
    virtual void start() const = 0;
};
```

2. محصولات ملموس (Concrete Products)

```
// درب بنز
class MercedesDoor : public Door {
public:
    void describe() const override {
        std::cout << "Mercedes Door: Luxurious and wide." << std::endl;
    }
    void open() { std::cout << "Shhhhhhhhhhh" << std::endl; }
};

// موتور بنز
class MercedesEngine : public Engine {
public:
    void start() const override {
        std::cout << "Guurrrruuuuumm, Mercedes Engine: V8, smooth ignition." <<
std::endl;
    }
};

// درب تویوتا
class ToyotaDoor : public Door {
public:
    void describe() const override {
        std::cout << "Toyota Door: Practical and lightweight." << std::endl;
    }
    void open() { std::cout << "zzzzzzzzzzzzzzzz" << std::endl; }
};

// موتور تویوتا
class ToyotaEngine : public Engine {
public:
    void start() const override {
        std::cout << "Toyota Engine: Efficient 4-cylinder." << std::endl;
    }
};
```

3. کارخانه‌ی انتزاعی (Abstract Factory)

```
// Abstract Factory
class CarFactory {
public:
    virtual ~CarFactory() = default;
    virtual std::unique_ptr<Door> createDoor() const = 0;
    virtual std::unique_ptr<Engine> createEngine() const = 0;
};
```

4. کارخانه‌های ملموس subclass کارخانه abstract (Concrete Factories)

```
// کارخانه بنز
class MercedesFactory : public CarFactory {
public:
    std::unique_ptr<Door> createDoor() const override {
        return std::make_unique<MercedesDoor>();
    }
};
```

```

        std::unique_ptr<Engine> createEngine() const override {
            return std::make_unique<MercedesEngine>();
        }
};
// کارخانه تویوتا
class ToyotaFactory : public CarFactory {
public:
    std::unique_ptr<Door> createDoor() const override {
        return std::make_unique<ToyotaDoor>();
    }
    std::unique_ptr<Engine> createEngine() const override {
        return std::make_unique<ToyotaEngine>();
    }
};

```

5. کلاس سرویس یا کارخانه‌ی مرکزی (Client)

```

class CarAssembler {
private:
    std::unique_ptr<Door> door;
    std::unique_ptr<Engine> engine;
public:
    CarAssembler(const CarFactory& factory) {
        door = factory.createDoor();
        engine = factory.createEngine();
    }
    void assemble() const {
        std::cout << "Assembling car with:\n";
        door->describe();
        engine->start();
        std::cout << "✅ Assembly complete.\n\n";
    }
};

```

6. تابع اصلی (Main)

```

#include <iostream>
#include <memory>
int main() {
    // تولید ماشین بنز
    MercedesFactory mercedesFactory;
    CarAssembler benzCar(mercedesFactory);
    benzCar.assemble();
    // تولید ماشین تویوتا
    ToyotaFactory toyotaFactory;
    CarAssembler toyotaCar(toyotaFactory);
    toyotaCar.assemble();
    return 0;
}

```

```
Assembling car with:
Mercedes Door: Luxurious and wide.
Mercedes Engine: V8, smooth ignition.
☒ Assembly complete.
Assembling car with:
Toyota Door: Practical and lightweight.
Toyota Engine: Efficient 4-cylinder.
☒ Assembly complete.
```

مزایای پترن Abstract Factory

- یکپارچگی محصولات: تضمین می‌کند که اجزای یک شی با یکدیگر سازگار باشند.
- استقلال از جزئیات پیاده‌سازی: کد کلاینت به کلاس‌های ملموس وابسته نیست.
- قابلیت افزودن خانواده‌های جدید: برای افزودن یک برند جدید (مثل BMW)، کافی است یک Concrete Factory جدید بنویسیم؛ بدون نیاز به تغییر کد کلاینت.
- همگنی رابط کاربری: تمام خانواده‌ها از یک رابط استاندارد پیروی می‌کنند.

معایب

- پیچیدگی اضافی: برای سیستم‌های کوچک، ممکن است اضافه باشد.
- تغییرات در محصولات: اگر بخواهیم یک قطعه محصول جدید (مثل تایر) اضافه کنیم، باید تمام Factoryها را تغییر دهیم (چون رابط کارخانه تغییر می‌کند).

جمع‌بندی

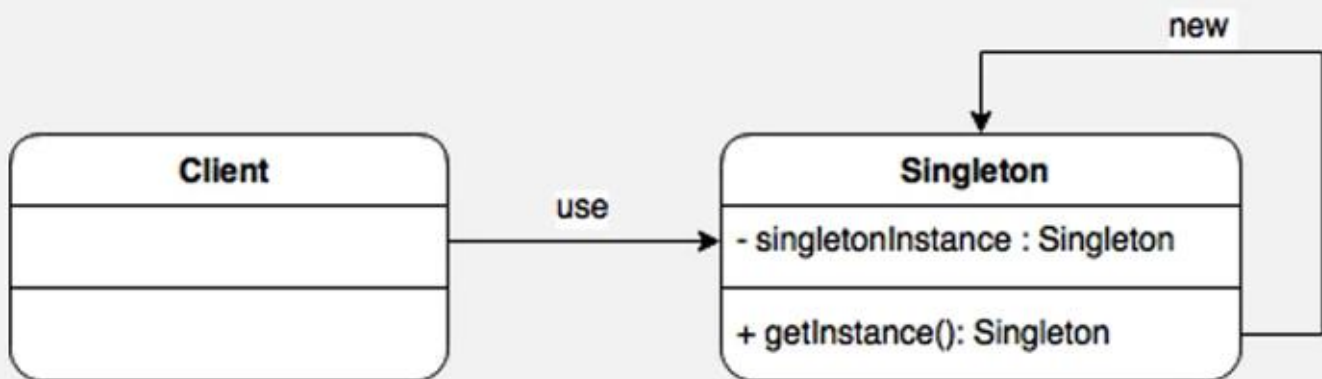
پترن Abstract Factory زمانی بسیار مفید است که با خانواده‌های مرتبط از اشیاء سروکار داریم و می‌خواهیم هرگز ترکیب نادرستی از آن‌ها رخ ندهد. در مثال ماشین‌سازی، این پترن تضمین کرد که هرگز موتور بنز با درِ تویوتا ترکیب نشود!

این پترن، پایه‌ای محکم برای معماری‌های انعطاف‌پذیر و قابل تست فراهم می‌کند و به‌ویژه در سیستم‌هایی با پیکربندی‌های مختلف (مثل تم‌ها، پلتفرم‌ها، یا برندها) بسیار کاربردی است.

از مثالهای دیگر در مورد این الگو، اقلام مختلف UI نرم افزارها در تم‌های مختلف است. مثلاً تمام دکمه‌ها و فونت نوشته‌ها و ... در تم مشکی به شکل خاص خود است و در تم سفید به شکل خاص خود است و ... و پس می‌توان کارخانه‌ای برای اقلام UI در تم مشکی ساخت و کارخانه دیگری مخصوص اقلام تم سفید و ...! مثال دیگرش بازیهای ماشین جنگی است که خودروها، راننده‌ها و تجهیزات و سلاح‌های خاص خود را دارند و یا بازیهای شامل کاراکترهای با آپشن‌های خاص که هر کاراکتر آپشن‌ها و سلاحها و ابزار خاص خود را دارد! مثال دیگر، بازی Mortal Kombat است که هر کاراکتر، لباس‌ها، فنون، ابزار و Finisherهای خاص خود را دارد و در حین ساختن هر کاراکتر نباید فنون و مدل و صدای کاراکتر دیگری را به وی نسبت دهیم!



Singleton pattern – Class diagram



این الگو، یک الگوی طراحی است که در بسیاری موارد به جهت اینکه خود باعث دردسر می شود، به آنتی پترن تبدیل می شود و دردسرساز می شود و پیشنهاد می شود تا حد امکان از آن استفاده نشود چون مثلاً در برخی موارد، تست را مشکل می کند و ...

این پترن در زمانی به کار می رود که ما نخواهیم به صورت ارادی هیچ شیی از یک کلاس دلخواه بسازیم جز فقط و فقط یک شی به صورت یکتا!

برای عدم امکان ساختن شی، متد `Constructor` را `private` تعریف می کنیم تا از بیرون از کلاس نتوانیم آنرا صدا بزنیم و یک متد استاتیک `public` به نام `Instance` یا `getInstance` می سازیم که تنها شی ساخته شده از کلاس را به ما می دهد و می توانیم متد

دیگری به نام `deleteInstance` بسازیم تا به وسیله آن، تک شی ساخته شده را پاک کنیم! یک اشاره گر هم به نام `_instance` از نوع خود کلاس به صورت `private static` با مقدار اولیه `nullptr` تعریف می کنیم و در واقع با `getInstance` امتحان می کنیم که اگر اشاره گر شی یکتای ما یعنی `_instance` مقدار `nullptr` داشته باشد، یعنی شی مد نظر ساخته نشده است و آنرا می سازیم و اگر قبلا ساخته شده بوده که هیچ و نهائیتا در هر دو صورت، شی ساخته شده را در قالب اشاره گر، به بیرون از `getInstance` پاس می دهیم. متد `deleteInstance` هم بررسی می کند که اگر اشاره گر شی مد نظر `nullptr` نباشد، آنرا `delete` و نهائیتا `nullptr` کند. همچنین در ادامه می توانید `copy constructor` و `move constructor` و حتی `assign operator` و ... را هم `delete` کنید تا کاربر فقط و فقط بتواند به طور مستقیم با خود متد `getInstance` کار کند و همه مسیرهای کپی کردن شی ساخته شده را کور کنید!

```
class SingletonDemo {
public:
    static SingletonDemo * getInstance(params) {
        if (!_instance) {
            _instance = new SingletonDemo(params);
        }
        return _instance;
    }
    static void deleteInstance() {
        if (_instance) {
            delete _instance;
            _instance = nullptr;
        }
    }

    SingletonDemo(const SingletonDemo&) = delete;

    SingletonDemo& operator=(const SingletonDemo&) = delete;

    SingletonDemo(SingletonDemo&&) = delete;

    SingletonDemo& operator=(SingletonDemo&&) = delete;

private:
    static SingletonDemo* _instance;
}
SingletonDemo* SingletonDemo::_instance = nullptr; //initialization of static
```

 **چالش:** اگر در یک برنامه، چند ترد با شی سینگلتون کار کنند و متد `getInstance` را همزمان فرا بخوانند، آیا می توانند در ساخت شی واحد دچار مشکل شوند؟ مثلا آیا امکان دارد حداقل دو ترد شرط `if _instance == nullptr` را تایید کنند و سپس هر کدام شی خود را بسازند و نهائیتا یکی از دو شی نامعتبر باشد (چون دیگری آنرا `Override` کرده است)!

 **پاسخ:** از `C++` نسخه 11 به بعد، ساخت اشیای `static` خود به خود اتمیک و `Thread Safe` است و قاعدتا نیازی به استفاده از ابزار دیگر نیست ولی در نسخه های قبلی `C++` بله! امکان دارد دو شی موازی به صورت همروند در ساخت شی سینگلتون در یک آن واحد، دوبار آنرا بسازند! یک نکته دیگر اینکه از `C++` نسخه 17 به بعد، با تعریف `static member` به صورت `inline static` می توانید مستقیما مقدار پیش فرض را در فایل هدر `.h` یا `.hpp`، تنظیم کنید و دیگر نیاز ندارید حتما آنرا در فایل `.cpp`، اصطلاحات `Initialize` کنید و مقادیر پیش فرض آنرا تنظیم کنید!

 **راه حل کلی؟** استفاده از `mutex` در کلاس سینگلتون!

```
#include <mutex>
class Singleton {
private:
    static Singleton* instance;
    static mutable std::mutex mtx;
    //میوتیبل باعث می شود حتی در اشیای کانست از این کلاس، قابل تغییر باشد
    Singleton() = default;
public:
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;
    static Singleton* getInstance() {
        std::lock_guard<std::mutex> lock(mtx);
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }
};
Singleton* Singleton::instance = nullptr;
std::mutex Singleton::mtx;
```

روش دیگر مدیریت چند نخه: استفاده از `std::call_once` است که در قسمت مربوط به **Lazy Initialization** توضیح داده ایم!

✗ ایرادات Singleton پترن چیست؟

- **حالت Global** و وابستگی های پنهان ایجاد می کند!
- **تست پذیری:** تست پذیری پایین - Mock کردن و تست کردن سخت است! معمولا در تست ها چند شی از یک کلاس بوجود می آورند و آنها را در موارد گوناگون تست می کنند، حالی که سینگلتون فقط یک شی دارد!
- **tight coupling:** کدها به هم وابسته می شوند!
- **مشکلات همروندی:** در محیط های چندنخی
- **نشکننده:** نقض اصل تک مسئولیت و وارونگی وابستگی
- **سینگلتون با Constructor پیچیده:** کلاس سینگلتونی که **Constructor** پیچیده با چند آرگمان ورودی دارد، اگر متد **GetInstance** داشته باشد، در صورت نبود **instance** مجبور است یک **Instance** جدید بسازد! پس مجبور است **GetInstance** را با تمام آرگمان های ورودی کانستراکتورش بگیرد تا در نبود **Instance**، شی جدید را با کمک آنها بسازد! ولی تصور کنید هر جا که به **Instance** یکتای سینگلتون نیاز داریم، مجبور باشیم آنها را با تمام پارامترها صدا بزنیم! مشکل اصلی آنجایی نمایان می شود که برخی آرگمان های ورودی، اشیایی از کلاسهای دیگر درون کلاس خاصی باشند و به آنها در همه جای کد که نیاز به شی یکتای سینگلتون داریم، دسترسی نداشته باشیم!

💡 راهکار نوشتن کلاس سینگلتون با سازنده پیچیده:

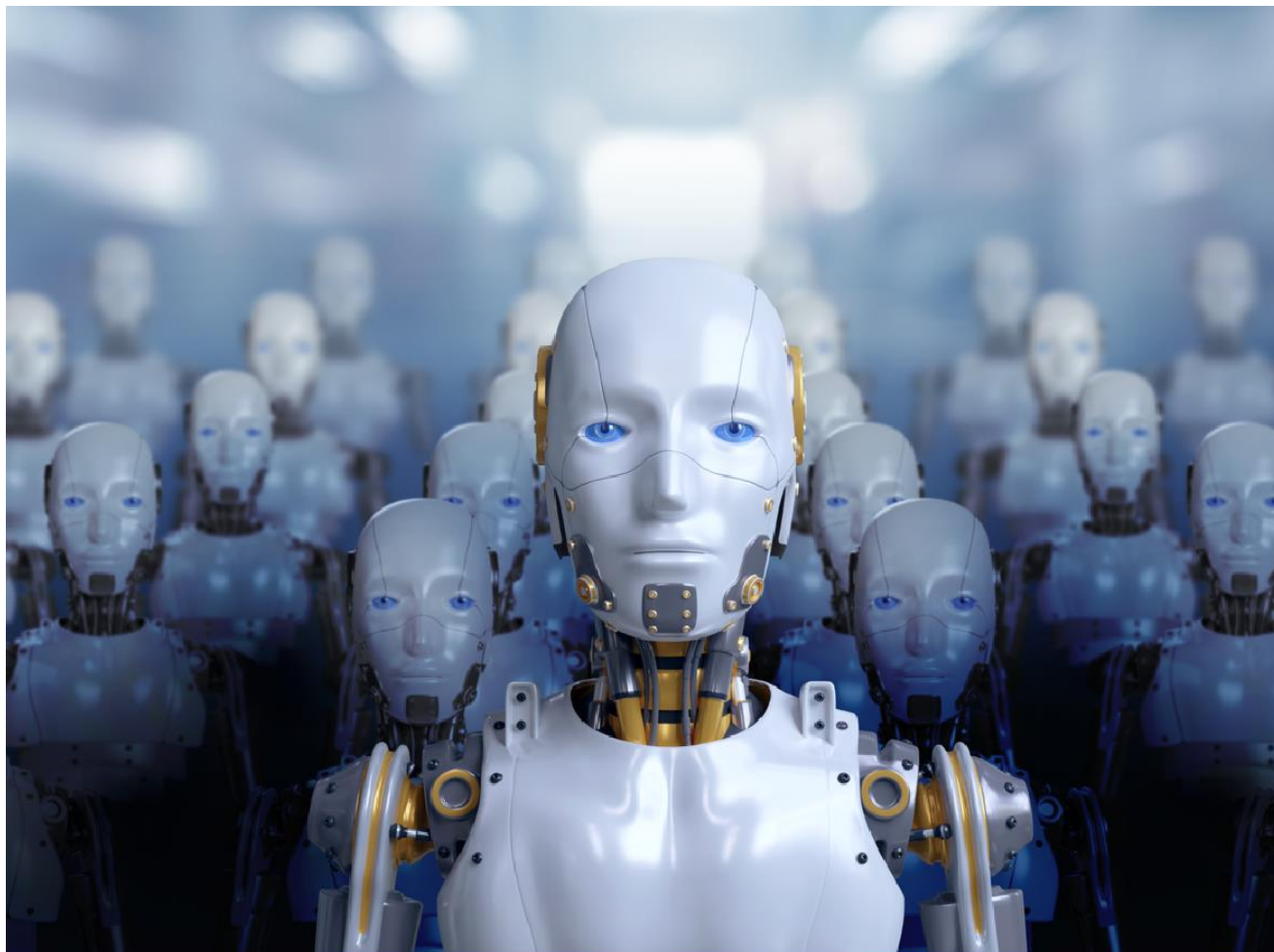
سوال: اگر کلاس سینگلتونی داشته باشید که کانستراکتور آن مثلا 5 آرگمان ورودی داشته باشد، آیا باید متد **GetInstance** آن، هر 5 آرگمان را دریافت کند تا در صورت **null** بودن **instance**، آنرا با هر 5 آرگمان ورودی اش بسازد؟

💡 **پاسخ:** می توان برای چنین شرایطی یک متد جداگانه دیگر **مشابه** **getInstance** به نام **buildInstance** ساخت که این متد جدید 5 آرگمان دریافت می کند و در صورت نبود **instance** آنرا می سازد و آدرس شی یکتا را درون **instance** می ریزد و نهایتا با استفاده از متد **getInstance** بدون آرگمان، **instance** یکتا را برمی گردانید! در اینجا ساخت شی جدید (در صورت عدم وجود) با **buildInstance** و دسترسی به آن **getInstance** و پاک کردن آن هم با **deleteInstance** است! پس برای ساخت شی جدید، اول باید شی قبلی را با **deleteInstance** پاک کنید و سپس با **buildInstance** شی جدید را بسازید و با

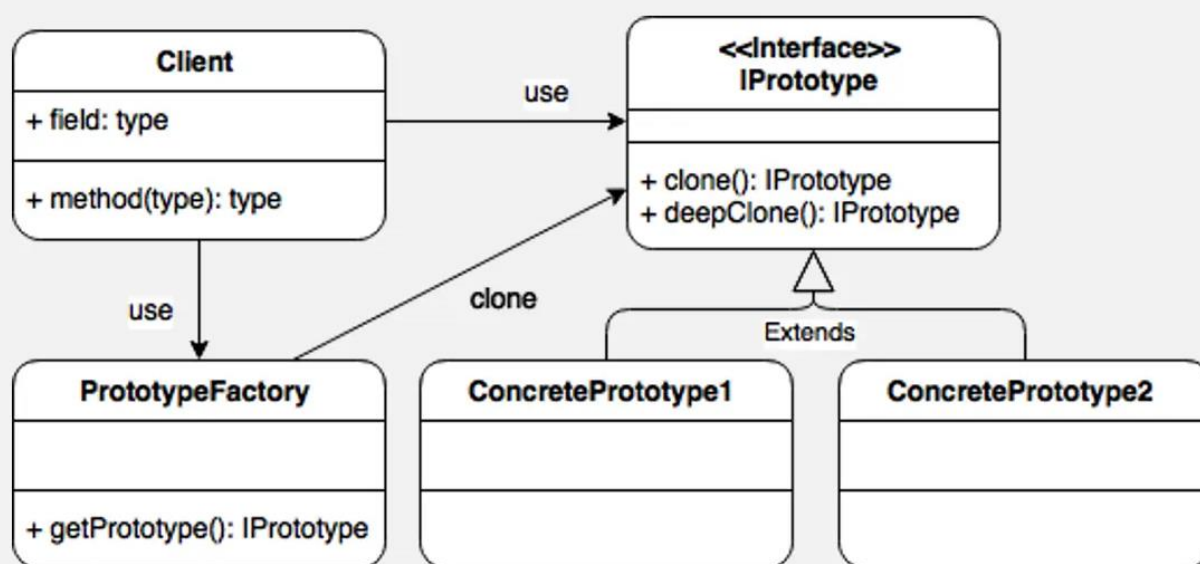
getInstance آنرا Return کنید و اگر buildInstance را Call نکرده باشید، طبیعتاً getInstance مقدار nullptr را بر می گرداند!

```
class Singleton {
public:
    static bool BuildInstance(int a, double b, ClassC* c, ClassD* d, bool e ) {
        if (_instance) return false;
        _instance = new Singleton(a,b,c,d,e);
        return true;
    }
    static Singleton* GetInstance() {
        return _instance;
    }
    static bool DeleteInstance() {
        if (_instance) delete _instance;
        _instance = nullptr;
    }
private:
    inline static Singleton* _instance = nullptr;
    Singleton(int a, double b, ClassC* c, ClassD* d, bool e) {
        //...
    }
    //...
};
```

- **Prototype**: پترنی از نوع Creational است و اشیا جدید را با کپی کردن نمونه‌های موجود و نمونه‌سازی از روی نمونه اولیه ایجاد می‌کند!



Prototype pattern – Class diagram



در الگوی **Prototype**، ما به جای ساخت اشیا از صفر (با سازنده‌های پر از پارامتر)، یک نمونه اولیه (**Prototype Object**) می‌سازیم، و هر بار که نیاز به شیء جدید داریم، آن را **Clone** یا کپی می‌کنیم!

به عبارت دیگر: در این الگو، به جای استفاده از **new** برای ساختن اشیای جدید، از نمونه‌ای از همان نوع، با متد **clone()** استفاده می‌کنیم.

💡 **مسئله‌ای که Prototype حل می‌کند!** فرض کنید در بازی مزرعه‌داری (**Farm Game**)، انواع حیوانات داریم:

- گاو (Cow)
- گوسفند (Sheep)
- مرغ (Chicken)

هر کدام از این حیوانات ویژگی‌های خاصی دارند:

- رنگ
- وزن
- سن
- صدای مخصوص
- رفتارها (مانند تولید شیر یا پشم)

حال اگر بخواهیم هر بار با سازنده‌ها (**constructor**) اشیای مختلف با ویژگی‌های متفاوت بسازیم، هم کد تکراری زیاد می‌شود، هم ممکن است اشیای مشابه را بارها بسازیم. از طرفی فرض کنید هر کدام از این موارد باید تکستچر و متریال خود را از ریسورس‌های بیرونی همچون وب سایت یا فایل لوکال لود کنند! هم مصرف **IO** و هم زمان لود هر آیتم بالا می‌رود! حال فرض کنید هزاران آیتم از هر کدام از انواع این اقلام چه بلایی سر پرفرمنس و زمان و کندی برنامه به بار می‌آورد!

💡 **راه حل؟** ایجاد یک نمونه اولیه از هر نوع حیوان (**Prototype**) و سپس کپی کردن (**Clone**) آن در زمان نیاز.

🔧 ساختار کلی الگوی Prototype

```
//classDiagram
class Prototype {
    <<interface>>
    +clone() : Prototype*
}
class ConcretePrototype1 {
    +clone() : Prototype*
}
class ConcretePrototype2 {
    +clone() : Prototype*
}
Prototype <|-- ConcretePrototype1
Prototype <|-- ConcretePrototype2
```

```
#include <iostream>
#include <string>
#include <memory>

class Animal {
public:
    virtual ~Animal() = default;
    virtual std::unique_ptr<Animal> clone() const = 0;
    virtual void speak() const = 0;
    virtual void info() const {
        std::cout << "نام: " << name << " | رنگ: " << color << " | سن: " << age << std::endl;
    }
    void setName(const std::string& n) { name = n; }
    void setColor(const std::string& c) { color = c; }
    void setAge(int a) { age = a; }
protected:
    std::string name;
    std::string color;
    int age{};
};
```

در اینجا متد clone() وظیفه ساختن یک کپی دقیق (Deep Copy) از شیء را دارد.

```
class Cow : public Animal {
public:
    Cow() { name = "گاو"; color = "قهوه‌ای"; age = 2; }
    std::unique_ptr<Animal> clone() const override {
        return std::make_unique<Cow>(*this); // استفاده از کپی سازنده
    }
    void speak() const override {
        std::cout << "مووو" << std::endl;
    }
};

class Sheep : public Animal {
public:
    Sheep() { name = "گوسفند"; color = "سفید"; age = 1; }
    std::unique_ptr<Animal> clone() const override {
        return std::make_unique<Sheep>(*this);
    }
    void speak() const override {
        std::cout << "بعبع" << std::endl;
    }
};
```

هر حیوان پیاده‌سازی خاص خودش از متد clone() دارد.


```
int main() {
    // ایجاد نمونه‌های اولیه (Prototype)
    std::unique_ptr<Animal> cowPrototype = std::make_unique<Cow>();
    std::unique_ptr<Animal> sheepPrototype = std::make_unique<Sheep>();
    // کلون کردن از روی Prototype
    auto cow1 = cowPrototype->clone();
    cow1->setColor("سیاه");
    cow1->setAge(3);
    auto cow2 = cowPrototype->clone();
    cow2->setColor("سفید");
    auto sheep1 = sheepPrototype->clone();
    sheep1->setAge(2);
    // نمایش اطلاعات حیوانات
    std::cout << "🐮 گاوها:" << std::endl;
    cow1->info();
    cow2->info();
    std::cout << "\n🐑 گوسفندان:" << std::endl;
    sheep1->info();
    std::cout << "\n🗣 صداهای:" << std::endl;
    cow1->speak();
    sheep1->speak();

    return 0;
}
```

خروجی برنامه 

```
🐮 گاوها:
نام: گاو | رنگ: سیاه | سن: 3 سال
نام: گاو | رنگ: سفید | سن: 2 سال
🐑 گوسفندان:
نام: گوسفند | رنگ: سفید | سن: 2 سال
🗣 صداهای:
!مووو
!بعبع
```

Prototype در زبان‌های پویا (JavaScript / ActionScript): در زبان‌هایی مثل JavaScript، مفهوم Prototype 

در ذات سیستم شیء‌گرایی وجود دارد. مثلاً:

```
let animal = {
    color: "brown",
    sound: "moo",
    speak() {
        console.log(this.sound);
    }
};
let cow = Object.create(animal); // کلون از پروتوتایپ
cow.speak(); // moo
cow.sound = "mooo"; // حالا تغییر داده‌ایم
cow.speak(); // mooo
animal.speak(); // moo ← حالا دیگر cow از پروتوتایپ انیمال جدا شده
```

در واقع در این زبان‌ها، اشیای جدید از نمونه‌های قبلی ساخته می‌شوند و تا زمانی که ویژگی خاصی را در نمونه جدید **override** نکرده باشیم، مقدار از والد گرفته می‌شود! مقادیر به روز شده والد در فرزندان **clone** شده ای که هنوز از والد جدا نشده اند هم منعکس می شوند ولی به محض جدا شدن یک شی از **prototype** اش به واسطه تغییر در مقادیر **attribute** هایش دیگر از والد تبعیت نمی کنند!

مزایا و نکات مهم Prototype

مزایا:

1. ساخت سریع اشیا بدون نیاز به سازنده‌های سنگین.
2. جدا شدن ساخت از کلاس‌های پیچیده.
3. قابل توسعه می‌توان Prototype جدید اضافه کرد بدون تغییر کدهای قدیمی.
4. انعطاف بالا برای Clone کردن اشیای پیچیده با ترکیب داده‌های مختلف.
5. عدم نیاز به لودینگ‌های سنگین: با یک بار لود شدن یک شی یا جزئی از یک شی، می‌توان از همان به تعداد دلخواه کپی گرفت!

نکات و معایب:

1. نیاز به پیاده‌سازی دقیق **clone()** بخصوص در **Deep Copy** یعنی اگر در کلاس **prototype**، اشاره گرهایی وجود داشته باشد، لازم است به جای اشاره اشاره گرهای شی **clone** شده به همان نقطه از حافظه شی اولیه، آنها را به اشیای **clone** شده جدید اشاره بدهیم! مثلاً اگر قرار است ماشینی را **clone** کنیم که شامل یک اشاره گر به یک راننده و یک اشاره گر به یک موتور ماشین است، لازم است یک راننده و یک موتور جدید **new** کنیم و اشاره گر شی **clone** شده را به موتور و راننده جدید اشاره دهیم! وگرنه راننده و موتور خودروی **prototype** و خودروی **clone** شده یکی خواهد بود!!! در قسمت آخر معرفی این الگو به توضیحات تکمیلی در این باره می‌پردازیم!
2. اگر شیء دارای اشاره گرها یا منابع پویا باشد، باید مدیریت حافظه با دقت انجام شود.
3. در زبان‌هایی مثل **C++**، باید از **smart pointer** ها (**unique_ptr**, **shared_ptr**) برای جلوگیری از **memory leak** استفاده شود.

 **نکته حرفه‌ای:** در پروژه‌های واقعی، **Prototype** معمولاً همراه با **Factory** یا **Registry** استفاده می‌شود. مثلاً:

```
class AnimalRegistry {
public:
    void addPrototype(const std::string& type, std::unique_ptr<Animal> prototype) {
        prototypes[type] = std::move(prototype);
    }
    std::unique_ptr<Animal> create(const std::string& type) const {
        return prototypes.at(type)->clone();
    }
private:
    std::unordered_map<std::string, std::unique_ptr<Animal>> prototypes;
};

// مثال استفاده
AnimalRegistry registry;
registry.addPrototype("cow", std::make_unique<Cow>());
registry.addPrototype("sheep", std::make_unique<Sheep>());
auto newCow = registry.create("cow");
newCow->setColor("مشکی");
```

این ساختار، Prototype Pattern را به یک Factory مبتنی بر نمونه‌ها تبدیل می‌کند که بسیار قدرتمند است.

 **تفاوت Deep Copy و Shallow Copy:** تفاوت اصلی میان کپی سطحی (Shallow Copy) و کپی عمیق (Deep Copy) در نحوه برخورد با ارجاع‌ها (Reference) و اشاره‌گرها است:

- در **Shallow Copy**: تنها خود شیء اصلی و مقدار متغیرهای اولیه (Primitive Types) کپی می‌شوند، اما برای متغیرهای ارجاعی (اشاره‌گرها)، فقط آدرس حافظه آن‌ها کپی می‌شود. در نتیجه، شیء اصلی و شیء کلون‌شده هر دو به یک نقطه یکسان از حافظه اشاره می‌کنند.
- در **Deep Copy**: علاوه بر خود شیء اصلی، تمام اشیاء وابسته که توسط اشاره‌گرها به آن‌ها ارجاع داده شده نیز به صورت مستقل و کاملاً جدید در حافظه ساخته (new) می‌شوند تا دو شیء کاملاً مجزا و بی‌ارتباط با یکدیگر داشته باشیم.

 **مثال متفاوتی از کلاس Car به همراه Driver و Engine:** فرض کنید کلاس Car دارای دو اشاره‌گر است:

1. Driver* driver که اشاره‌گر به راننده ماشین است!
2. Engine* engine که اشاره‌گر به موتور ماشین است!

```
#include <iostream>
#include <string>
// =====
// ۱. کلاس‌های پایه: راننده (Driver) و موتور (Engine)
// =====
class Driver {
public:
    std::string name;

    Driver(const std::string& name) : name(name) {}
    // متد کلون برای ایجاد یک نمونه مستقل جدید (کپی عمیق)
    Driver* clone() const {
        return new Driver(*this);
    }
};

class Engine {
public:
    int horsepower;

    Engine(int hp) : horsepower(hp) {}
    // متد کلون برای ایجاد یک نمونه مستقل جدید (کپی عمیق)
    Engine* clone() const {
        return new Engine(*this);
    }
};

// =====
// ۲. کلاس اصلی: خودرو (Car) حاوی اشاره‌گر به Driver و Engine
// =====
class Car {
public:
    std::string model;
    Driver* driver; // اشاره‌گر به راننده
    Engine* engine; // اشاره‌گر به موتور
```

```

Car(const std::string& model, Driver* driver, Engine* engine)
    : model(model), driver(driver), engine(engine) {}

~Car() {
    //Memory Leak ازجمله برای جلوگیری
    delete driver;
    delete engine;
}

// -----
// روش اول - Shallow Copy - کپی سطحی
// -----
// در این روش فقط آدرسهای حافظه کپی میشوند
// شیء جدید به همان راننده و موتور اشاره میکند که شیء اصلی اشاره میکرد
// عارضه جانبی: تغییر در راننده/موتور ماشین جدید، روی ماشین اصلی هم اثر میگذارد
Car* shallowCopy() const {
    return new Car(this->model + " (Shallow)", this->driver, this->engine);
}

// -----
// روش دوم - Deep Copy - کپی عمیق - الگوی (Prototype)
// -----
// در این روش اشیای وابسته Driver و Engine به صورت مستقل new میشوند
// اشاره گرهای شیء جدید به نمونه های کاملاً تازه اختصاص می یابند
// نتیجه: دو ماشین کاملاً مجزا در حافظه تشکیل میشوند و هیچ تداخلی ندارند
Car* deepCopy() const {
    Driver* newDriver = this->driver->clone(); // new کردن راننده جدید
    Engine* newEngine = this->engine->clone(); // new کردن موتور جدید

    return new Car(this->model + " (Deep)", newDriver, newEngine);
}

void displayInfo() const {
    std::cout << "Model: " << model
                << " | Driver: " << driver->name
                << " | Engine HP: " << engine->horsepower << std::endl;
}

};

// =====
// نحوه استفاده و بررسی تفاوت Shallow Copy و Deep Copy
// =====

int main() {
    //ایجاد شیء اولیه (Prototype)
    Driver* originalDriver = new Driver("Ali");
    Engine* originalEngine = new Engine(200);
    Car* originalCar = new Car("Tesla Model S", originalDriver, originalEngine);

    std::cout << "--- Original Car ---" << std::endl;
    originalCar->displayInfo();
    // -----
    // تست Shallow Copy
    // -----
    Car* shallowCar = originalCar->shallowCopy();
    //تغییر نام راننده در شیء کلون شده (Shallow)
    shallowCar->driver->name = "Reza";

    std::cout << "\n--- After modifying driver name in Shallow Copy ---" << std::endl;
    std::cout << "Original Car: " << originalCar->displayInfo();
    //نام راننده ماشین اصلی هم تغییر کرد (Ali -> Reza)
    std::cout << "Shallow Car: " << shallowCar->displayInfo();
}

```

```
// -----
// Deep Copy تست
// -----
Car* deepCar = originalCar->deepCopy();
// (Deep) تغییر نام راننده و قدرت موتور در شیء کلون شده
deepCar->driver->name = "Hassan";
deepCar->engine->horsepower = 350;

std::cout << "\n--- After modifying driver and engine in Deep Copy ---" << std::endl;
std::cout << "Original Car: " << originalCar->displayInfo();
// ماشین اصلی بدون تغییر باقی ماند
std::cout << "Deep Car: " << deepCar->displayInfo();
// ماشین جدید راننده و موتور مستقل خود را دارد
// پاکسازی حافظه
delete originalCar;
// توجه shallowCar: به همان driver/engine اشاره می‌کند که پاک شد
// اما deepCar دارای حافظه مستقل است و باید جداگانه پاک شود
delete deepCar;

return 0;
}
```

جمع‌بندی 

مفهوم	توضیح
هدف	ساخت اشیای جدید از روی یک نمونه اولیه به جای سازنده
مکان استفاده	وقتی ایجاد شیء جدید هزینه‌بر یا پیچیده است
کلید اصلی	متد clone() که نمونه جدید را برمی‌گرداند
شباهت با JS Prototype	رفتار ارث‌بری مبتنی بر نمونه اولیه تا زمان override شدن مقادیر
ترکیب رایج	اغلب همراه با Factory یا Registry استفاده می‌شود

 انواع ساخت نمونه Copy از یک شی ساخته شده از یک کلاس یا استراکت:

1- Dereference کردن از یک اشاره‌گری که به یک شی از یک کلاس یا استراکت اشاره می‌کند!

```
MyClass* main_object = new MyClass();
```

```
auto copy_version = *main_object;
```

2- استفاده از کانستراکتور Copy کلاس / استراکتی که می‌خواهیم نمونه‌کپی آنرا بسازیم (به شرطی که کانستراکتور کپی آن وجود داشته باشد!)

```
Auto main_object = new MyClass{.....};
```

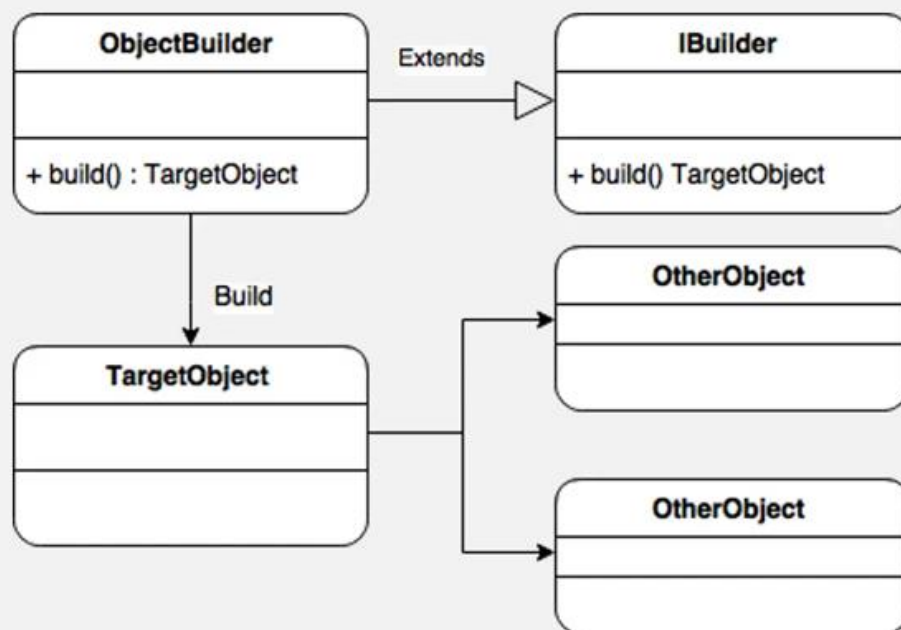
```
main_object ->setPropertyX(x);
```

```
auto copy_version = new MyClass{main_object}; //duplicate using copy constructor
```

برای داپلیکیت کردن یک شی برای استفاده در الگوی Prototype می‌توان از هر یک از دو روش بالا بهره جست!



Builder pattern – Class diagram



این الگو، ساخت اشیا پیچیده را گام به گام مدیریت می‌کند! در حقیقت این الگو در جایی به کار می‌رود که یک سیستم متشکل از چند زیرسیستم جزئی تر تشکیل می‌شود که هر کدام از زیر سیستم‌ها، خود کلاس و سیستمی جداگانه هستند و اگر بخواهیم انواع سیستم اصلی را با

انواع متنوع زیر سیستم های گوناگون پدید آوریم، مجبور می شویم با این الگوی کارآمد و تست شده برای این نوع مسایل استفاده کنیم! در واقع الگوی **Builder** یکی از الگوهای ساختاری (**Creational Pattern**) است که هدف آن ساخت مرحله به مرحله اشیاء پیچیده است، بدون اینکه نیاز باشد سازندهی (**Constructor**) کلاس، تعداد زیادی پارامتر بگیرد.

به زبان ساده، **Builder** الگوی مقابله با "سازندهای تلسکوپ" است! سازنده تلسکوپ **Constructor** ای است که برای ساخت یک شیء باید دهها پارامتر را به سازنده بدهید، که نه خوانا است و نه قابل نگهداری.

ایده اصلی

در بسیاری از برنامه‌ها، اشیاء ساده نیستند!

- مثلاً فرض کنید می‌خواهید یک ساندویچ بسازید!
- یا مثلاً می‌خواهید یک منوی گرافیکی (**Menu**) برای نرم‌افزار طراحی کنید.

هر کدام از این‌ها چندین ویژگی اختیاری دارند (مثل نوع نان، سس، حاشیه، رنگ، جهت و غیره).

اگر بخواهیم این ویژگی‌ها را مستقیماً با سازنده تنظیم کنیم، تعداد پارامترها زیاد می‌شود و خوانایی از بین می‌رود:

```
Menu menu("main", "Main Menu", {"Option1", "Option2"}, true, 2);
```

در نگاه اول ساده است، اما وقتی پارامترها زیاد می‌شوند، به سختی می‌توان فهمید هر عدد یا مقدار برای چیست.

تعریف الگوی Builder

الگوی **Builder** ساخت اشیاء پیچیده را به چند مرحله تقسیم می‌کند، به طوری که بتوان این مراحل را مستقل از یکدیگر اجرا کرد. این الگو اجازه می‌دهد:

- ساخت اشیاء به صورت گام به گام انجام شود.
- بتوان نمایش‌ها یا انواع مختلفی از یک شیء را با همان کد **Builder** تولید کرد.

کد ساخت (**Builder**) از منطق استفاده **Director** یا **Client** جدا شود.

مثال اول: ساخت منو (Menu)

فرض کنید در یک برنامه رابط کاربری (UI)، می‌خواهیم منوهای بسازیم که می‌توانند شامل عنوان، گزینه‌ها، حاشیه و جهت افقی یا عمودی باشند.

 **نسخه بدون Build: Constructor** ساده استفاده کنیم، باید همه چیز را یکجا بدهیم:

```
class Menu {
public:
    Menu(std::string id,
          std::string title,
          std::vector<std::string> options,
          bool horizontal,
          int border)
    : mId(id), mTitle(title), mOptions(options),
      mHorizontal(horizontal), mBorder(border) {}
};
```



```
private:
    std::string mId;
    std::string mTitle;
    std::vector<std::string> mOptions;
    bool mHorizontal;
    int mBorder;
};
```

اگر تعداد ویژگی‌ها بیشتر شود، باید چندین سازنده‌ی مختلف بسازیم یا مقادیر پیش‌فرض بدهیم، که باز هم ناخوشایند است.

🌱 **نسخه با Builder ساده:** حالا ببینیم Builder چطور به ما کمک می‌کند تا کد تمیزتری داشته باشیم!

```
class Menu {
public:
    static Menu create(std::string id) {
        return Menu{id};
    }

    Menu& withTitle(std::string title) {
        _title = title;
        return *this;
    }
    Menu& withBorder(int pixels) {
        _border = pixels;
        return *this;
    }

    Menu& addOption(std::string option) {
        _options.push_back(option);
        return *this;
    }
    Menu& horizontal() {
        _horizontal = true;
        return *this;
    }
    Menu& vertical() {
        _horizontal = false;
        return *this;
    }
    void show() const {
        std::cout << "Menu: " << _title << " (border: " << _border << ")\n";
    }
private:
    std::string _id;
    std::string _title;
    std::vector<std::string> _options;
    bool _horizontal{false};
    int _border{0};
    Menu(std::string id) : _id(id) {}
};
```

استفاده: 🛠️

```
const auto mainMenu = Menu::create("main")
    .withTitle("Main Menu")
    .addOption("Option 1")
    .addOption("Option 2")
    .horizontal()
    .withBorder(2);
mainMenu.show();
```

مزایا:

- کد خوانا تر است!
- می‌توانیم فقط ویژگی‌های موردنیاز را تنظیم کنیم
- به راحتی می‌توانیم متدها را زنجیره کنیم (Method Chaining)

ساختر کلی الگوی Builder

نقش	توضیح
Builder (Abstract Interface)	تعریف متدهای ساخت بخش‌های مختلف شیء
ConcreteBuilder	پیاده‌سازی متدهای ساخت و تولید محصول نهایی
Director	کنترل ترتیب ساخت مراحل
Product	شیء نهایی که در پایان ساخته می‌شود

مثال دوم: ساخت اتومبیل با Builder کلاسیک

فرض کنید می‌خواهیم اشیاء مختلف از نوع **Car** بسازیم که شامل موتور، بدنه، رنگ، تایر و نوع صندلی هستند.

می‌توانیم از ساختار Builder کلاسیک با نقش‌های مشخص استفاده کنیم:

- **Product** → Car
- **Builder (Interface)** → CarBuilder
- **ConcreteBuilder** → SportsCarBuilder, SUVBuilder
- **Director** → کلاس مسئول تعیین ترتیب ساخت!

تعریف کلاس‌ها

```
class Car {
public:
    void SetEngine(const std::string& e) { engine = e; }
    void SetBody(const std::string& b) { body = b; }
    void SetColor(const std::string& c) { color = c; }
    void Show() const {
        std::cout << "Car built with: " << body << ", "
                    << engine << ", color: " << color << "\n";
    }
private:
    std::string engine, body, color;
};
```

```
class CarBuilder {
public:
    virtual ~CarBuilder() = default;
    virtual void BuildEngine() = 0;
    virtual void BuildBody() = 0;
    virtual void Paint() = 0;
    virtual std::unique_ptr<Car> GetCar() = 0;
};
```

: Concrete Builder پیاده‌سازی 

```
class SportsCarBuilder : public CarBuilder {
private:
    std::unique_ptr<Car> car;
public:
    SportsCarBuilder() { car = std::make_unique<Car>(); }
    void BuildEngine() override { car->SetEngine("V8 Turbo"); }
    void BuildBody() override { car->SetBody("Sport Coupe"); }
    void Paint() override { car->SetColor("Red"); }
    std::unique_ptr<Car> GetCar() override { return std::move(car); }
};
```

:Director 

```
class Director {
private:
    CarBuilder* builder;
public:
    void SetBuilder(CarBuilder* b) { builder = b; }
    std::unique_ptr<Car> Build() {
        builder->BuildBody();
        builder->BuildEngine();
        builder->Paint();
        return builder->GetCar();
    }
};
```

:main استفاده در 

```
int main() {
    Director director;
    SportsCarBuilder sportsBuilder;
    director.SetBuilder(&sportsBuilder);
    auto car = director.Build();
    car->Show();
}
```

مثال ساده‌تر: ساخت ساندویچ (Sandwich Builder) 

در این مثال می‌خواهیم انواع مختلف ساندویچ (همبرگر، هات‌داگ، BLT) را با ساختار Builder تولید کنیم.

```
#include <iostream>
#include <vector>
#include <utility> // برای std::move

enum class Bread { Toast, HotdogBun, BurgerBun }; // نان ساندویچ
enum class Protein { Beef, Sausage, Bacon }; // گوشت - پروتئین
enum class Sauce { Ketchup, Mayo, Mustard }; // سس
```

```

class Sandwich {
public:
    Bread bread;
    Protein protein;
    std::vector<Sauce> sauces;
    void Show() const {
        std::cout << "Sandwich ready with " << sauces.size() << " sauces!\n";
    }
};

class SandwichBuilder {
public:
    virtual ~SandwichBuilder() = default;
    virtual SandwichBuilder& AddBread() = 0;
    virtual SandwichBuilder& AddProtein() = 0;
    virtual SandwichBuilder& AddSauce() = 0;
    Sandwich GetResult() {
        // برای جلوگیری از کپی شدن بیهوده وکتور سسها std::move استفاده از
        Sandwich result = std::move(_sandwich);
        Reset(); // آماده سازی بیلدر برای ساخت ساندویچ بعدی
        return result;
    }
protected:
    Sandwich _sandwich;

    void Reset() {
        _sandwich = Sandwich{}; // مقداردهی اولیه مجدد به حالت پیشفرض
    }
};

class BurgerBuilder : public SandwichBuilder {
public:
    SandwichBuilder& AddBread() override {
        _sandwich.bread = Bread::BurgerBun;
        return *this; // بازگشت به خود بیلدر
    }
    SandwichBuilder& AddProtein() override {
        _sandwich.protein = Protein::Beef;
        return *this;
    }
    SandwichBuilder& AddSauce() override {
        _sandwich.sauces.push_back(Sauce::Ketchup);
        return *this;
    }
};

class Director {
public:
    Sandwich Make(SandwichBuilder& b) {
        // حالا زنجیره سازی به درستی روی خود Builder انجام می شود
        b.AddBread()
        .AddProtein()
        .AddSauce();
        return b.GetResult();
    }
};

```

استفاده: 

```

int main() {
    Director d;
    BurgerBuilder builder;
    Sandwich s = d.Make(builder);
    s.Show();
}

```

مزایا:

- ساخت اشیاء پیچیده به صورت مرحله به مرحله
- افزایش خوانایی و کاهش خطای انسانی
- جداسازی منطق ساخت از منطق استفاده
- انعطاف پذیری در تولید اشیاء متفاوت با همان مراحل ساخت

موارد استفاده:

- ساخت فرم‌های UI با گزینه‌های زیاد
- ساخت گزارش‌ها یا فایل‌های PDF مرحله‌ای
- ساخت موجودیت‌های بازی (مثلاً شخصیت با سلاح و لباس متنوع)
- ساخت مدل‌های سه‌بعدی با تنظیمات متعدد

جمع‌بندی

الگوی Builder زمانی به کار می‌رود که:

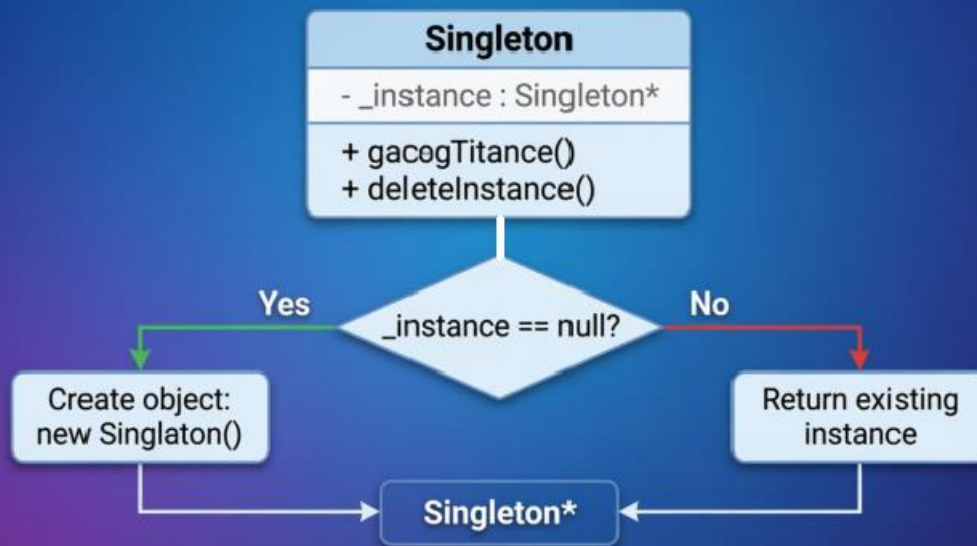
- شیء شما ویژگی‌های زیادی دارد (بعضی اجباری و بعضی اختیاری)
- سازنده‌های زیاد باعث پیچیدگی و تکرار می‌شوند
- نیاز دارید اشیاء مختلف را با فرآیند ساخت مشابه بسازید

در واقع، **Builder** یک راه تمیز برای مدیریت پیچیدگی در ساخت اشیاء است و به جای سازنده‌ای با ده پارامتر، می‌گویید:

```
auto burger = BurgerBuilder()  
    .AddBread()  
    .AddProtein()  
    .AddSauce()  
    .GetResult();
```



Singleton Pattern with Lazy Initialization



الگوی Lazy Initialization در C++ یعنی مقداردهی هوشمند آن هم فقط هنگام نیاز

«تنبلی» به عنوان فضیلت! در دنیای نرم افزار، منابع محدود هستند! حافظه، زمان پردازش، پهنای باند، و حتی باتری! بنابراین، ایجاد یک شیء پیچیده قبل از نیاز واقعی، می تواند منجر به هدررفت منابع، کاهش عملکرد و رفتار غیرپیش بینی شده شود!

الگوی (Lazy Initialization مقداردهی تنبل) این اصل را پیاده می کند!

“یک شیء را تا زمانی که واقعاً مورد نیاز نباشد، ایجاد یا مقداردهی نکن!”

این الگو به‌ویژه در C++ قدرتمند است، چون به ما امکان می‌دهد تا از کنترل دقیق روی زمان ساخت (construction time) و مدیریت دستی حافظه استفاده کنیم.

🔑 موارد کاربرد رایج

- منابع سنگین: اتصال به دیتابیس، بارگذاری فایل‌های بزرگ، ایجاد شبکه‌های عصبی.
- اشیای تنها: (Singleton) معمولاً تنها زمانی نیاز است که اولین بار درخواست شود.
- محاسبات گران‌قیمت قابل کش شدن: مثلاً محاسبه ماتریس‌های معکوس یا مسیرهای بهینه یا هر محاسبه سنگینی که می‌توان کش کرد!
- وابستگی‌های شرطی: شیء‌ای که فقط در حالت خاصی مورد استفاده قرار می‌گیرد.

🌱 پیاده‌سازی‌های مختلف در C++: نسخه ساده غیر thread-safe

```
class HeavyResource {
public:
    HeavyResource() { /* ساخت گران‌قیمت */ }
    void use() { /* ... */ }
};

class LazyHolder {
private:
    mutable std::unique_ptr<HeavyResource> _resource;
public:
    const HeavyResource& getResource() const {
        if (!_resource) {
            _resource = std::make_unique<HeavyResource>();
        }
        return *_resource;
    }
};
```

✅ مزایا: ساده و خوانا

❌ عیب: در محیط چنددخی (multithreaded) ایمن نیست.

🌱 نسخه thread-safe با mutex (Double-Checked Locking) :

```
class LazyHolder {
private:
    mutable std::unique_ptr<HeavyResource> _resource;
    mutable std::mutex _mtx;
public:
    const HeavyResource& getResource() const {
        بررسی اولیه بدون قفل، کلا قفل‌تردی در چک کردن همیشگی سربار دارد //
        std::lock_guard<std::mutex> lock(_mtx);
        بررسی نهایی با قفل //
        if (!_resource) {
            _resource = std::make_unique<HeavyResource>();
        }
        return *_resource;
    }
};
```


✔ مزایا: کاهش overhead قفل در دسترسی‌های مکرر

✗ معایب: پیچیده تر است، و در C++ قدیمی ممکن است به دلیل reorder کامپایلر مشکل داشته باشد!

💡 نکته: از C++11 به بعد، std::mutex و مدل حافظه جدید، Double-Checked Locking را ایمن می‌کند به شرطی که از std::atomic با نوع‌های thread-safe استفاده شود.

✂ نسخه ایده‌آل: استفاده از std::call_once و std::once_flag

این روش، ساده‌ترین و ایمن‌ترین راه برای Lazy Initialization thread-safe در C++11 به بعد است:

```
class LazyHolder {
private:
    mutable std::unique_ptr<HeavyResource> _resource;
    mutable std::once_flag _initFlag;
public:
    const HeavyResource& getResource() const {
        std::call_once(_initFlag, [this]() {
            _resource = std::make_unique<HeavyResource>();
        });
        return *_resource;
    }
};
```

✔ مزایا:

- کاملاً thread-safe
- بدون overhead قفل در دسترسی‌های بعدی
- خوانا و کوتاه
- توسط استاندارد تضمین شده

✂ نسخه فوق‌العاده ساده: متغیر محلی (static برای توابع)

اگر نیاز شما یک شیء تنها (Singleton) یا منبع جهانی است، ساده‌ترین راه:

```
const HeavyResource& getResource() {
    static HeavyResource _instance; // Lazy + thread-safe در C++11 به بعد
    return _instance;
}
```

✔ مزیت: از C++11 به بعد، ساخت متغیر static محلی در تابع، به‌طور خودکار thread-safe است!

✔ مزیت: بدون نیاز به mutex، once_flag یا کلاس اضافه

✗ عیب: فقط برای یک نمونه جهانی قابل استفاده است (نه برای اعضای کلاس)

Singleton در الگوی Lazy Initialization

ترکیب Singleton با Lazy Initialization بسیار رایج است:

```
class DatabaseConnection {
public:
    static DatabaseConnection& getInstance() {
        static DatabaseConnection _instance; // Lazy + thread-safe
        return _instance;
    }
    DatabaseConnection(const DatabaseConnection&) = delete;
    DatabaseConnection& operator=(const DatabaseConnection&) = delete;
private:
    DatabaseConnection() { /* اتصال واقعی به دیتابیس */ }
};
```

این پیاده‌سازی پیشنهاد اصلی اکثر کتاب‌های مدرن C++ است .

مزایا

- صرفه‌جویی در منابع (حافظه، زمان، انرژی)
- بهبود زمان راه‌اندازی برنامه
- امکان مدیریت وابستگی‌های شرطی

معایب

- پیچیدگی کد در نسخه‌های thread-safe
- ممکن است تأخیر غیرمنتظره در اولین استفاده ایجاد کند

جمع‌بندی:

Lazy Initialization تنها یک «تکنیک بهینه‌سازی» نیست بلکه یک فلسفه طراحی هوشمند است: کاری را تا زمانی که واقعاً لازم نباشد، انجام نده! در C++، با ابزارهای مدرن `std::unique_ptr`، `std::call_once` و همچنین `static` محلی داخل یک تابع، می‌توان این الگو را هم ایمن، هم ساده و هم کارآمد پیاده‌سازی کرد .

* یادآوری نهایی:

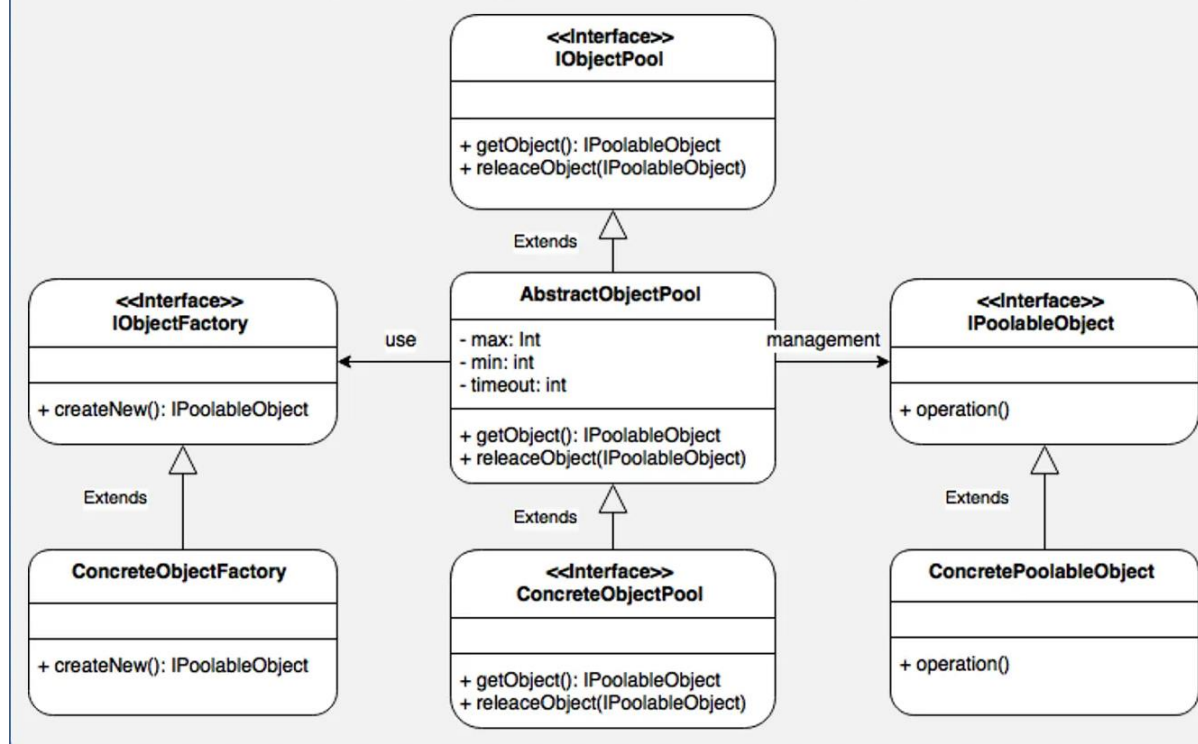
اگر می‌خواهید یک شیء را فقط یک بار به صورت تنبل بسازید، ساده‌ترین و بهترین راه در C++11 همیشه این است :

```
static YourType instance;
```

○ **Object Pool** : نوعی از الگوهای ساختاری Creational است! بازیابی، نه بازسازی!



Object Pool pattern – Class diagram



چرا هر بار شیء بسازیم؟

در بسیاری از برنامه‌ها به‌ویژه در بازی‌ها، شبکه، پردازش واقعیت مجازی یا سیستم‌های **real-time** با اشیایی روبرو هستیم که:

- **ساخت و نابودی‌شان گران‌قیمت** است (مثلاً اتصال شبکه، بافر گرافیکی، اشیاء فیزیکی در موتور بازی، شیء ارتباط با دیتابیس).
 - **به‌صورت موقت استفاده می‌شوند** و سپس دور انداخته می‌شوند.
 - **به‌سرعت و به تعداد زیاد** نیاز است (مثلاً گلوله‌های تیراندازی باز یا بسته‌های شبکه). به بیان بهتر آهنگ ساخت و نابودیشان بالاست!
- اگر هر بار یک شیء جدید از **heap** تخصیص دهیم (**new**) و سپس آن را آزاد کنیم (**delete**)، با دو مشکل جدی مواجه می‌شویم:
1. **هزینه زمانی بالا** در ساخت/نابودی (مثلاً فراخوانی سیستم‌عامل برای حافظه).
 2. **تکه‌تکه شدن حافظه (memory fragmentation)** که در سیستم‌های **real-time** بحرانی است.

 **الگوی Object Pool** این مشکل را با یک ایده ساده حل می‌کند:

یک مخزن از اشیاء از پیش ساخته‌شده نگه دار! وقتی نیاز بود قرض بگیر! وقتی کارت تمام شد برگردان! اگر تعداد اشیای داخل مخزن کمتر از تعداد نیاز بود، اشیای جدید بساز و به مخزن اضافه کن!

? چه زمانی از Object Pool استفاده کنیم؟

- زمانی که ساخت/نابودی شیء گران‌قیمت است.
- زمانی که اشیاء کوتاه عمر هستند و به‌سرعت می‌آیند و می‌روند.
- زمانی که نیاز به عملکرد قابل پیش‌بینی داریم (مثلاً در بازی‌ها یا سیستم‌های **real-time**).
- زمانی که تعداد همزمان اشیاء محدود است (مثلاً حداکثر ۱۰۰ اتصال شبکه).

 **ساختار کلی الگو:** یک **ObjectPool<T>** معمولاً شامل موارد زیر است:

مخزن یا Storage	معمولاً یک std::queue یا std::stack از اشیاء یا اشاره گر ها دارد!
تابع acquire	یک شیء از استخر بر می‌گرداند (اگر استخر خالی بود، یکی می‌سازد)
تابع release (obj)	شیء را به استخر باز می‌گرداند تا دوباره مورد استفاده قرار گیرد!
تابع Reset (obj) برای اولیه سازی	قبل از بازگشت شیء باید به حالت اولیه Reset شود

 **پیاده‌سازی thread-safe در C++17 به بعد:**

در اینجا یک پیاده‌سازی کامل، انعطاف‌پذیر و **thread-safe** از **ObjectPool** آورده شده است!

```

#include <memory>
#include <queue>
#include <mutex>
#include <functional>
template<typename T>

class ObjectPool {
public:
    // سازنده: اختیاراً استخر را با تعدادی شیء اولیه پر می‌کند
    // در ورودی هم سائز اولیه دادیم و هم تابع فکتوری بدون ورودی
    explicit ObjectPool(std::size_t initialSize = 0,
        std::function<std::unique_ptr<T>()> factory = []() {
            return std::make_unique<T>();
        })
        : factory_(std::move(factory)) {
        for (std::size_t i = 0; i < initialSize; ++i) {
            pool_.push(factory_());
        }
    }
    // دریافت یک شیء از استخر
    std::unique_ptr<T> acquire() {
        std::lock_guard<std::mutex> lock(mutex_);
        if (pool_.empty()) {
            return factory_();
        }
        auto obj = std::move(pool_.front());
        pool_.pop();
        return obj;
    }
    void release(std::unique_ptr<T> obj) { // بازگرداندن شیء به استخر
        if (obj) {
            obj->reset();
            // فرض: شیء متد reset() دارد
            std::lock_guard<std::mutex> lock(mutex_);
            pool_.push(std::move(obj));
        }
    }
    std::size_t available() const { // تعداد اشیاء آزاد در استخر
        std::lock_guard<std::mutex> lock(mutex_);
        return pool_.size();
    }
private:
    mutable std::mutex mutex_;
    std::queue<std::unique_ptr<T>> pool_;
    std::function<std::unique_ptr<T>()> factory_;
};

```

نکته: فرض شده که کلاس T یک متد `reset()` دارد که آن را به حالت اولیه بازمی‌گرداند. اگر چنین متدی ندارد، می‌توانید از یک تابع بازگرداننده (`reinitializer`) به عنوان پارامتر سازنده استفاده کنید

مثال کاربردی: استخر گلوله‌ها در یک بازی 2 بعدی:

```

class Bullet {
private:
    float start_x = 0, start_y = 0, end_x = 0, end_y = 0;
    bool active = false;
public:

```

```

void fire(float startX, float startY, float endX, float endY) {
    start_x = startX;
    start_y = startY;
    end_x = endX;
    end_y = endY;
    active = true;
}

void update() {
    if (active) {
        if (start_x < end_x)
            start_x += 0.1;
        else if (start_x > end_x)
            start_x -= 0.1;
        if (start_y < end_y)
            start_y += 0.1;
        else if (start_y > end_y)
            start_y -= 0.1;

        if (CheckBulletHitToTarget()) active = false;
    }
}

bool isActive() const { return active; }
// بازگرداندن به حالت اولیه
void reset() {
    start_x = start_y = end_x = end_y = 0;
    active = false;
}
};

int main() { // استفاده
    ObjectPool<Bullet> bulletPool(50); // استخر با ۵۰ گلوله اولیه

    auto bullet = bulletPool.acquire(); // دریافت یک گلوله از استخر
    bullet->fire(50, 50, 200, 300); // 300 و 200 نقطه به 50 و 50 نقطه
    // بعد از استفاده
    bulletPool.release(std::move(bullet)); // به استخر بازمی‌گردد
    return 0;
}

```

❁ نسخه پیشرفته بدون نیاز به **reset()**: اگر کلاس شما **reset()** ندارد، می‌توانیم از کپسوله‌سازی منطق بازگردانی استفاده کنیم:

```

template<typename T>
class ObjectPool {
    std::function<void(T&)> reinitializer_ = [](T& obj) { /* default */ };
    // ...
public:
    void release(std::unique_ptr<T> obj) {
        if (obj) {
            reinitializer_(*obj); // منطق بازگردانی قابل تنظیم
            std::lock_guard<std::mutex> lock(mutex_);
            pool_.push(std::move(obj));
        }
    }
};

```

مزایا و معایب:

مزایا

- کاهش overhead ساخت/نابودی
- جلوگیری از تکه‌تکه شدن حافظه
- قابلیت پیش‌بینی عملکرد critical (در real-time)
- کنترل بر حداکثر منابع مصرفی

معایب

- پیچیدگی کد (مدیریت حالت شیء)
- مصرف حافظه همیشگی (حتی وقتی شیء استفاده نمی‌شود)
- خطر استفاده نادرست (مثلاً فراموش کردن release)
- نیاز به thread-safety در محیط‌های چنددخی

Object Pool در مقابل الگوهای دیگر:

وقتی در استفاده از منابع داخل pool به کمبود منابع می‌خوریم، از Factory می‌توان برای ساخت شیء جدید می‌توان استفاده کرد!

در Factory می‌توان از prototype نمونه‌های جدید ساخت!

می‌توان Pool‌های مختلف در زمینه‌های مختلف مثل Database Connection یا Bullet ها یا ... داشت ولی هر کدام از انواع این Pool را می‌توان به صورت singleton به خدمت گرفت! در اینجا می‌توان الگوی Flyweight (قسمت‌های بعدی) را هم به خدمت گرفت! یعنی در جایی که می‌توان از یک الگوی مشترک برای 2 یا بیشتر اشیاء استفاده کرد، می‌توان از Map ها برای نگه داشتن اشیاءی مشترک استفاده کرد تا هر نوع شیء بر اساس type‌اش از داده هم‌ارز خود استفاده کند!

بهترین روش‌ها (Best Practices):

1. همیشه شیء را قبل از بازگشت به استخر reset کنید، در غیر این صورت، داده‌های قدیمی ممکن است باعث bug شوند.
2. حداکثر اندازه استخر را محدود کنید، برای جلوگیری از نشت حافظه.
3. در سیستم‌های چنددخی، حتماً از mutex یا atomic استفاده کنید!
4. استخر را در آغاز برنامه پیش‌پر کنید تا از تاخیر در اولین استفاده، جلوگیری کنید!
5. از smart pointer (unique_ptr) استفاده کنید، البته برای جلوگیری از نشت حافظه در صورت استثنا.

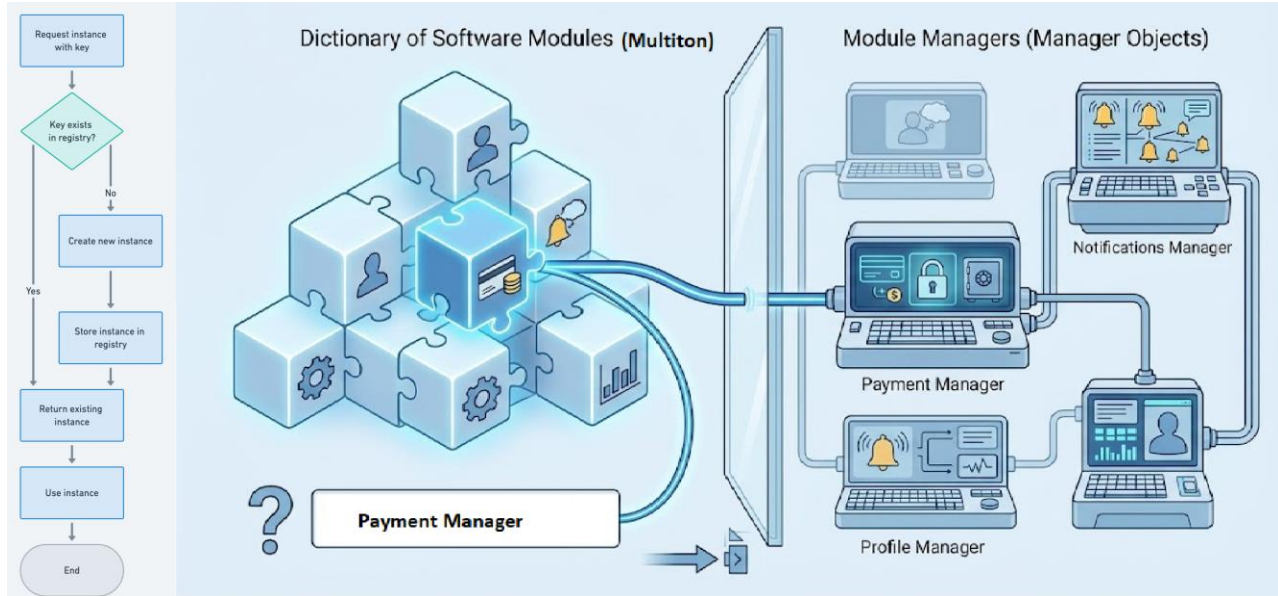
 جمع بندی: Object Pool یک الگوی قدرتمند برای بهینه‌سازی عملکرد و مدیریت منابع است!

در C++، با ترکیب RAII، Smart Pointer و ابزارهای همزمانی (mutex, lock_guard)، می‌توان یک استخر شیء ایمن و کارآمد و قابل نگهداری ساخت!

 یادآوری نهایی:

استخر شیء مثل یک کتابخانه است! قرض بگیر، استفاده کن، سپس برگردان تا دیگران هم بتوانند از آن لذت ببرند.

Multiton: نوعی از الگوهای ساختاری Creational است!



الگوی Multiton (تلفظ: مولتی تون، یا چندتنها) گسترشی از الگوی معروف Singleton (تک تنها) است. در حالی که Singleton تضمین می کند تنها یک نمونه (instance) از یک کلاس در طول برنامه وجود داشته باشد، Multiton به شما اجازه می دهد تعداد محدود و کنترل شده ای از نمونه ها را بر اساس یک کلید (key) ایجاد و مدیریت کنید.

این الگو در مواردی مفید است که نیاز دارید چند شیء از یک کلاس را به صورت کنترل شده و بازیابی پذیر نگه دارید مثلاً برای مدیریت دسته ای از منابع با نام یا شناسه مشخص.

در این الگو:

- برای هر Key، فقط یک نمونه ساخته می شود!
- اگر نمونه وجود داشت، همان برگردانده می شود!
- اگر وجود نداشت، ساخته و در Map ذخیره می شود!

```
auto* logger1 = Logger::getInstance("File");
auto* logger2 = Logger::getInstance("Console");
auto* logger3 = Logger::getInstance("File"); // همان logger1
```

🏠 ساختار کلی Multiton

در C++، Multiton معمولاً با استفاده از یک **map** داخلی مثلاً `std::unordered_map` پیاده سازی می شود که کلیدها را به نمونه های شیء مربوطه نگاشت می کند. سازنده (constructor) کلاس به صورت `private` یا `protected` تعریف می شود تا از ایجاد مستقیم نمونه جلوگیری شود.

```
#include <iostream>
#include <map>
#include <memory>
#include <string>

class Logger {
public:
    static Logger& getInstance(const std::string& key) {
        // اگر نمونه ای با این کلید نداریم، بسازیم
        if (instances.find(key) == instances.end()) {
            instances[key] = std::unique_ptr<Logger>(new Logger(key));
        }
        return *instances[key]; // نمونه را برگردان
    }
    void log(const std::string& msg) {
        std::cout << "[" << name << "]" << msg << std::endl;
    }
private:
    explicit Logger(const std::string& key) : name(key) {}
    std::string name;
    // استاتیک از کلید → مپ نمونه
    static std::unordered_map<std::string, std::unique_ptr<Logger>> instances;
};
// تعریف استاتیک در فایل cpp
std::unordered_map<std::string, std::unique_ptr<Logger>> Logger::instances;
```

استفاده: 🛠️

```
int main() {
    auto& fileLogger = Logger::getInstance("File");
    auto& consoleLogger = Logger::getInstance("Console");
    auto& fileLogger2 = Logger::getInstance("File");
    fileLogger.log("Saving data...");
    consoleLogger.log("Printing results...");
    fileLogger2.log("File logger reused!");
    return 0;
}
```

خروجی: 🖨️

```
[File] Saving data...

[Console] Printing results...

[File] File logger reused!
```

❖ پیاده سازی دیگر ساده از Multiton:

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <memory>
```

```

class Multiton {
public:
    // نوع کلید را می‌توان تغییر داد مثلاً int ، enum ، یا std::string
    using Key = std::string; // تعادل typedef
    // روشی برای دریافت یا ایجاد نمونه متناظر با یک کلید
    static Multiton& get_instance(const Key& key) {
        static std::unordered_map<Key, std::unique_ptr<Multiton>> instances;
        auto it = instances.find(key);
        if (it == instances.end()) {
            auto ptr = std::make_unique<Multiton>();
            ptr->id_ = key;
            instances[key] = std::move(ptr);
        }
        return *instances[key];
    }
    // جلوگیری از کپی و انتساب
    Multiton(const Multiton&) = delete;
    Multiton& operator=(const Multiton&) = delete;
    void do_something() const {
        std::cout << "Instance with ID: " << id_ << " is working.\n";
    }
private:
    Multiton() = default; // سازنده خصوصی
    std::string id_; // شناسه داخلی نمونه
};

```

استفاده: 

```

int main() {
    auto& db1 = Multiton::get_instance("database");
    auto& db2 = Multiton::get_instance("database");
    auto& cache = Multiton::get_instance("cache");

    db1.do_something(); // Instance with ID: database is working.
    db2.do_something(); // همان نمونه قبلی
    cache.do_something(); // Instance with ID: cache is working.
    // تأیید یکسان بودن نمونه‌های با کلید یکسان
    std::cout << (&db1 == &db2 ? "Same instance\n" : "Different instances\n");
    return 0;
}

```

Thread-Safe Version (C++11)

برای پروژه‌های بزرگ Multi Thread باید از ابزار Thread-Safe استفاده کنید. بهترین روش استفاده از mutex است!

```

#include <mutex>
class Logger {
public:
    static Logger& getInstance(const std::string& key) {
        std::lock_guard<std::mutex> lock(mtx);
        if (instances.find(key) == instances.end()) {
            instances[key] = std::unique_ptr<Logger>(new Logger(key));
        }
        return *instances[key];
    }
}

```

```

void log(const std::string& msg) {
    std::cout << "[" << name << "]" " << msg << std::endl;
}
private:
explicit Logger(const std::string& key) : name(key) {}
std::string name;
static std::map<std::string, std::unique_ptr<Logger>> instances;
static std::mutex mtx;
};
std::map<std::string, std::unique_ptr<Logger>> Logger::instances;
std::mutex Logger::mtx;

```

نسخه حرفه‌ای تر (Template-Based Multiton) اگر بخواهیم Multiton را برای هر کلاسی استفاده کنیم: 

```

#include <iostream>
#include <string>
#include <unordered_map>
#include <memory>
#include <shared_mutex>

template<typename Key, typename T>
class Multiton {
public:
    Multiton(const Multiton&) = delete; // حذف کانسٹرکتور کپی
    Multiton& operator=(const Multiton&) = delete; // حذف عملگر انتساب
    template<typename... Args>
    static std::shared_ptr<T> getPtr(const Key& key, Args&&... args) {
        {
            std::shared_lock<std::shared_mutex> readLock(mtx);
            auto it = instances.find(key);
            if (it != instances.end()) {
                return it->second;
            }
        }
        std::unique_lock<std::shared_mutex> writeLock(mtx);
        auto it = instances.find(key);
        if (it != instances.end()) {
            return it->second;
        }
        auto instance = std::make_shared<T>(std::forward<Args>(args)...);
        instances.emplace(key, instance);
        return instance;
    }

    template<typename... Args>
    static T& get(const Key& key, Args&&... args) {
        return *getPtr(key, std::forward<Args>(args)...);
    }

    static void remove(const Key& key) {
        std::unique_lock<std::shared_mutex> lock(mtx);
        instances.erase(key);
    }

    static bool contains(const Key& key) {
        std::shared_lock<std::shared_mutex> lock(mtx);
        return instances.find(key) != instances.end();
    }
}

```

```

static size_t count() {
    std::shared_lock<std::shared_mutex> lock(mtx);
    return instances.size();
}

static void clear() {
    std::unique_lock<std::shared_mutex> lock(mtx);
    instances.clear();
}
private:
    static std::unordered_map<Key, std::shared_ptr<T>> instances;
    static std::shared_mutex mtx;
};
template<typename Key, typename T>
std::unordered_map<Key, std::shared_ptr<T>> Multiton<Key, T>::instances;

template<typename Key, typename T>
std::shared_mutex Multiton<Key, T>::mtx;

```

استفاده: 

```

#include <iostream>
#include <string>
// --- Classes for Value (T) ---
class DatabaseConnection {
public:
    DatabaseConnection(const std::string& connStr) : connStr(connStr) {
        std::cout << "[DB Created] " << connStr << "\n";
    }
    void query(const std::string& q) {
        std::cout << "[DB Query on " << connStr << "]" << q << "\n";
    }
private:
    std::string connStr;
};

class Logger {
public:
    Logger(const std::string& name) : name(name) {
        std::cout << "[Logger Created] " << name << "\n";
    }
    void log(const std::string& msg) {
        std::cout << "[" << name << "]" << msg << "\n";
    }
private:
    std::string name;
};

class CacheManager {
public:
    CacheManager(int sizeMB) : sizeMB(sizeMB) {
        std::cout << "[Cache Created] Size: " << sizeMB << "MB\n";
    }
    void store() {
        std::cout << "[Cache] Storing data in " << sizeMB << "MB memory.\n";
    }
}

```

```

private:
    int sizeMB;
};

class Config {
public:
    Config(const std::string& env, int port) : env(env), port(port) {
        std::cout << "[Config Created] Env: " << env << ", Port: " << port << "\n";
    }
    void print() {
        std::cout << "Environment: " << env << " | Port: " << port << "\n";
    }
private:
    std::string env;
    int port;
};

int main() {
    std::cout << "=== 1. Multiton for DatabaseConnection (Key: string) ===\n";
    auto db1 = Multiton<std::string, DatabaseConnection>::getPtr("main_db",
"Server=localhost;DB=main;");
    auto db2 = Multiton<std::string, DatabaseConnection>::getPtr("main_db",
"Server=localhost;DB=main;");
    auto db3 = Multiton<std::string, DatabaseConnection>::getPtr("replica_db",
"Server=replica;DB=main;");
    db1->query("SELECT * FROM users;");

    std::cout << "Are db1 and db2 the same? " << (db1 == db2 ? "YES" : "NO") << "\n";
    std::cout << "Are db1 and db3 the same? " << (db1 == db3 ? "YES" : "NO") << "\n\n";

    std::cout << "=== 2. Multiton for Logger (Key: string) ===\n";
    auto& logger1 = Multiton<std::string, Logger>::get("info_log", "InfoLogger");
    auto& logger2 = Multiton<std::string, Logger>::get("error_log", "ErrorLogger");
    logger1.log("System started successfully.");
    logger2.log("Disk space is low!");
    std::cout << "Logger count: " << Multiton<std::string, Logger>::count() << "\n\n";

    std::cout << "=== 3. Multiton for CacheManager (Key: int) ===\n";
    auto cache1 = Multiton<int, CacheManager>::getPtr(1, 512);
    auto cache2 = Multiton<int, CacheManager>::getPtr(2, 1024);
    auto cache3 = Multiton<int, CacheManager>::getPtr(1, 999);
    cache1->store();
    cache3->store();
    std::cout << "CacheManager count: " << Multiton<int, CacheManager>::count() << "\n\n";

    std::cout << "=== 4. Multiton for Config (Key: string) ===\n";
    auto& devConfig = Multiton<std::string, Config>::get("development", "dev", 3000);
    auto& prodConfig = Multiton<std::string, Config>::get("production", "prod", 8080);
    auto& testConfig = Multiton<std::string, Config>::get("development", "dev", 9999);
    devConfig.print();
    prodConfig.print();
    testConfig.print();

    std::cout << "Are devConfig and testConfig the same? " << (&devConfig == &testConfig ?
"YES" : "NO") << "\n";
    std::cout << "Config count: " << Multiton<std::string, Config>::count() << "\n\n";

    return 0;
}

```



```
=== 1. Multiton for DatabaseConnection (Key: string) ===

[DB Created] Server=localhost;DB=main;

[DB Created] Server=replica;DB=main;

[DB Query on Server=localhost;DB=main;] SELECT * FROM users;

Are db1 and db2 the same? YES

Are db1 and db3 the same? NO

=== 2. Multiton for Logger (Key: string) ===

[Logger Created] InfoLogger

[Logger Created] ErrorLogger

[InfoLogger] System started successfully.

[ErrorLogger] Disk space is low!

Logger count: 2

=== 3. Multiton for CacheManager (Key: int) ===

[Cache Created] Size: 512MB

[Cache Created] Size: 1024MB

[Cache] Storing data in 512MB memory.

[Cache] Storing data in 512MB memory.

CacheManager count: 2

=== 4. Multiton for Config (Key: string) ===

[Config Created] Env: dev, Port: 3000

[Config Created] Env: prod, Port: 8080

Environment: dev | Port: 3000

Environment: prod | Port: 8080

Environment: dev | Port: 3000

Are devConfig and testConfig the same? YES

Config count: 2
```

❌ مشکلات و انتقادات به Multiton :

- اغلب global state ایجاد می‌کند!
- تست نویسی را سخت‌تر می‌کند!
- زیاد استفاده شود، سیستم پیچیده می‌شود!
- در DDD معمولاً توصیه نمی‌شود مگر برای منابع shared

✅ مزایا

- کنترل دقیق روی تعداد نمونه‌ها
- مدیریت ساده منابع مشترک
- جلوگیری از ایجاد ده‌ها instance اضافی
- API یکپارچه

✂ جایگزین‌های مدرن

در طراحی‌های مدرن (به‌ویژه با Dependency Injection یا Service Locator)، استفاده از Multiton کمتر توصیه می‌شود، چون می‌تواند کد را شکننده و غیرشفاف کند. با این حال، در موارد خاص (مثل مدیریت پول‌های ارتباطی با کانفیگ‌های ثابت) همچنان مفید است.

📌 جمع بندی

الگوی Multiton زمانی به درد می‌خورد که:

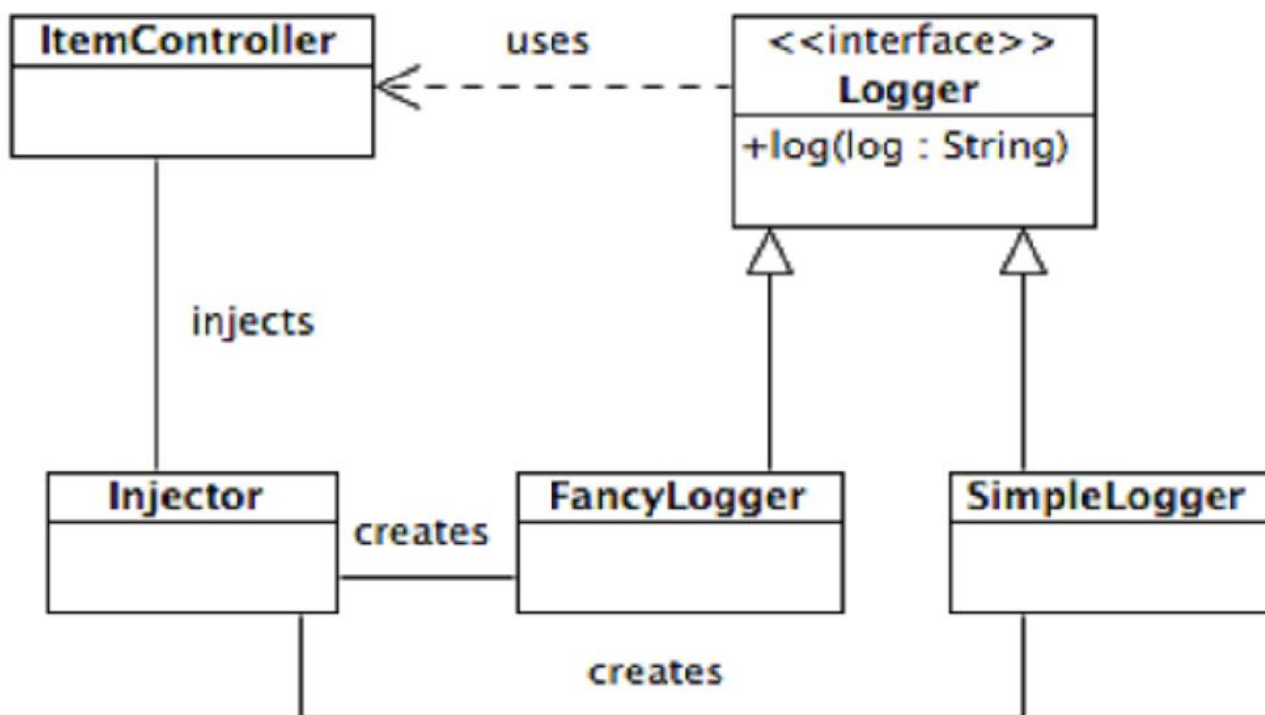
- نمی‌خواهیم تنها یک نمونه داشته باشیم مثل Singleton!
- اما نمی‌خواهیم بی‌نهایت نمونه هم آزادانه ساخته شود!
- بلکه نمونه‌ها باید بر اساس Key مدیریت شوند!

در C++ با استفاده از `std::map`, `unique_ptr` و `mutex` می‌توان یک Multiton کاملاً حرفه‌ای ساخت.

الگوی Multiton یک گسترش هوشمندانه از Singleton است که اجازه می‌دهد نمونه‌های کنترل‌شده‌ای از یک کلاس را بر اساس یک کلید مدیریت کنید. پیاده‌سازی آن در C++ با استفاده از `std::unordered_map` و محدود کردن دسترسی به سازنده امکان‌پذیر است. هرچند کاربردهای مفیدی دارد، باید با احتیاط و با آگاهی از چالش‌های مدیریت حافظه و تست‌پذیری استفاده شود.

○ الگوی **Dependency Injection (DI)**: این الگو به نوعی هم از نوع الگوهای **Creational** است و هم به نوعی از

نوع الگوهای **Structural**!



این اصل یکی از مهم‌ترین اصول طراحی نرم‌افزارهای مدرن، برای کاهش وابستگی (**Coupling**) بین اجزای سیستم است. هرچه کلاس‌ها و ماژول‌ها کمتر به پیاده‌سازی یکدیگر وابسته باشند، نگهداری، تست، توسعه و تغییر سیستم ساده‌تر خواهد بود! این الگو سنگ بنای ایجاد سیستم‌های ماژولار، قابل تست و قابل نگهداری است!

الگوی **Dependency Injection (DI)** یا «تزریق وابستگی» یکی از رایج‌ترین تکنیک‌ها برای پیاده‌سازی اصل **Inversion Of Control** یا **IoC** است و در معماری‌های مدرن مانند:

- Clean Architecture
- Domain-Driven Design (DDD)
- Hexagonal Architecture
- Microservices
- ASP.NET Core
- Spring Framework
- Angular

به‌صورت گسترده استفاده می‌شود.

Dependency چیست؟ Dependency یا وابستگی به این معناست که یک کلاس برای انجام وظیفه خود به کلاس دیگری نیاز دارد و آنرا Include یا import می‌کند!

مثال: 🎨

```
class Logger {
public:
    void Log(const std::string& msg) {
        std::cout << msg << std::endl;
    }
};

class UserService {
private:
    Logger logger;
public:
    void CreateUser() {
        logger.Log("User Created");
    }
};
```

در این مثال:

UserService ---> Logger

بنابراین UserService به Logger وابسته است.

❌ **مشکل وابستگی مستقیم:** فرض کنید بخواهیم Logger را تغییر دهیم و یکی از آیتم‌های زیر را جای آن قرار دهیم:

```
FileLogger
DatabaseLogger
CloudLogger
```

در طراحی بالا مجبوریم UserService را نیز تغییر دهیم. مثلاً:

```
class UserService {
private:
    FileLogger logger;
};
```

این موضوع اصل **Open/Closed Principle** در معماری **Solid** را نقض می‌کند.

💡 راه حل **Dependency Injection**: به جای اینکه کلاس خودش وابستگی را بسازد:

```
Logger logger;
```

وابستگی از بیرون به آن داده می‌شود.

```
UserService(Logger& logger);
```

🏠 اصل **Inversion of Control (IoC)**:

این اصل یک اصل کلی در طراحی نرم‌افزار است که می‌گوید کنترل جریان برنامه باید از کدهای سفارشی به یک فریم‌ورک یا کانتینر مرکزی مثل تابع **main** منتقل شود.

UserService ---> Logger

قبل از IoC، UserService مسئول ساخت Logger است.

Container ---> Logger ---> UserService

بعد از IoC مسئولیت ساخت وابستگی‌ها به بیرون منتقل می‌شود.

🏠 تعریف رسمی **Dependency Injection** چیست؟

Dependency Injection یک تکنیک برای فراهم کردن وابستگی‌های یک شی از خارج آن شی است، به جای اینکه خود شی وابستگی‌هایش را ایجاد کند. در واقع تزریق وابستگی الگویی است که در آن اشیاء (Objects) وابستگی‌های خود (یعنی اشیاء یا سرویس‌هایی که برای انجام کارشان به آن‌ها نیاز دارند) را خودشان نمی‌سازند، بلکه این وابستگی‌ها از بیرون به آن‌ها "تزریق" می‌شوند. به بیان بهتر تزریق وابستگی (DI) یک الگوی طراحی خاص برای پیاده‌سازی اصل IoC است. در DI، کنترل «ایجاد وابستگی‌ها» از کلاس مصرف‌کننده گرفته شده و به یک موجودیت بیرونی (مانند یک کانتینر DI یا تابعی مثل **main()** که کلاس‌ها را فراخوانی می‌کند) سپرده می‌شود.

به جای اینکه یک کلاس با استفاده از کلمه کلیدی **new** (یا معادل آن در زبان‌های دیگر) نمونه‌ای از یک وابستگی را ایجاد کند، آن وابستگی از طریق سازنده (Constructor)، متد (Setter) یا اینترفیس به کلاس داده می‌شود.

💡 چه وابستگی وارونه می‌شود؟

قبلاً کلاس سطح بالا مثل کلاس **Page** به کلاس سطح پایین مثل **ScrollBar** وابسته بود و آنرا **include** می‌کرد و از آن **Instance** می‌ساخت!

Page ---> ScrollBar

چرا که یک **Page** شامل یک **ScrollBar** بود!

اینک **Page** می‌تواند در درون خود یک **interface** به نام **IScrollBar** تعریف کند و توسط **Constructor** یا توسط یک **Setter** آنرا در قالب **IScrollBar** از بیرون دریافت می‌کند و از آن استفاده می‌کند! حال خود کلاس **ScrollBar** مجبور است به صورت معکوس کلاس **Page** را **include** کند و از **IScrollBar** به عبارتی **Implement** کند تا از بیرون از کلاس **Page** (داخل تابع **main**) یک نمونه از **ScrollBar** (به صورت اشاره گر) و یک نمونه **Page** بسازیم و شی ساخته شده از کلاس **ScrollBar** را به شی ساخته شده از کلاس **Page** وارد کنیم تا در قالب یک

ScrollBar ---> Page

IScrollBar در **Page** ست کنیم!

🌱 **روش بهتر:** می توان همه Interface ها یا Abstract class ها را در یک فایل همانند common.cpp , common.hpp تعریف کرد تا با

دسترسی به آن، همه کلاسها به همه اینترفیس ها به صورت مشترک دسترسی داشته باشند! یا در یک مسیر خاص، برای هر اینترفیس یک فایل .hpp, .cpp. مخصوص آنرا تعریف کنیم تا همه کلاسها بر اساس نیازمندی شان، اینترفیس خاص را include و سپس implement کنند!

همچنان که مشاهده می کنید اینجا دیگر وابستگی معکوس می شود!

Page ---> IScrollBar , ScrollBar ---> IScrollBar

✅ مزایای وابستگی به Interface به جای Class :

اگر آرایش اینترفیس را به درستی چیده باشید، به احتمال قوی ساختار ظاهری آن تغییر نخواهد کرد! بلکه شاید ماهیت درون توابع یک اسکرول بار به جهت ایرادات یا نیازهای جدید تغییر یابد! لذا اصل ساختار تغییر نمی یابد و شاید بعدا نیاز داشته باشید عملکرد توابع آنرا تغییر دهید تا اصلاحات مورد نیازتان نهایی شود و از انتشار تغییرات جلوگیری شود! از طرفی عدم تغییر اینترفیس و فقط تغییر در پیاده سازی کلاس زیرین همانند ScrollBar (فایل .cpp). یعنی عدم کامپایل دوباره Page و کلاس های بالادستی در تغییرات ScrollBar و این مساله در پروژه های بزرگ اهمیت دارد چون زمان کامپایل را کاهش می دهد! لذا در عین اینکه احتمال تغییر در کلاسها را کمتر کند، احتمال بالاتر بودن سرعت کامپایل را هم افزایش می دهد!

🎯 مثال بدون DI

```
class EmailSender {
public:
    void Send() {
        std::cout << "Email Sent";
    }
};

class UserService {
private:
    EmailSender sender;

public:
    void Register() {
        sender.Send();
    }
};
```

❌ مشکلات عدم استفاده از DI در این مثال:

1- وابستگی شدید **UserService**: UserService به شدت به کلاس EmailSender وابسته است!

(Tight Coupling). اگر بخواهیم به جای ایمیل، از SmsSender استفاده کنیم، باید کد UserService را تغییر دهیم (نقض اصل Open/Closed). به بیان بهتر برای ارسال SMS به جای Email باید SMSSender را در UserService به عبارتی include کنید و از آن نمونه سازی کنید!

UserService ---> EmailSender

2- **تست پذیری پایین (Poor Testability):** نوشتن تست واحد (Unit Test) برای این کلاس دشوار است، زیرا نمی‌توانیم به راحتی EmailSender واقعی را با یک نسخه جعلی (Mock) جایگزین کنیم تا از ارسال واقعی ایمیل در حین تست جلوگیری کنیم. بلکه مجبوریم خود کلاس را ویرایش کنیم و EmailSender را کامنت کنیم و به جایش نسخه Mock را تعریف و نمونه سازی کنیم تا تست را با آن پیش ببریم!

3- نقض اصل Open/Close Priciple و Solid در Dependency Inversion:

برای تغییر نوع ارسالگر پیام باید اصل کلاس را ویرایش کنیم (OCP) و کلاس UserService به کلاس EmailSender وابسته است!

مثال با DI 🍌

```
//Interface ابتدا
class IEmailSender
{
public:
    virtual ~IEmailSender() = default;

    virtual void Send() = 0;
};
//پایاده سازی

class EmailSender : public IEmailSender
{
public:
    void Send() override
    {
        std::cout << "Email Sent";
    }
};

//سرویس
class UserService
{
private:
    IEmailSender& _sender;
public:
    UserService(IEmailSender& sender)
        : _sender(sender)
    {}

    void Register()
    {
        _sender.Send();
    }
};

//استفاده
EmailSender sender;

UserService service(sender);

service.Register();
```

مزایای Dependency Injection

- 1- کاهش **Coupling** : به جای وابستگی به Concrete Class از نوع EmailSender، به Interface نوع EmailSender وابسته می‌شویم!
- 2- افزایش **Testability**: بدون DI تست سخت است ولی با DI:

```
class FakeEmailSender : public IEmailSender
{
public:
    bool called = false;
    void Send() override
    {
        called = true;
    }
};

FakeEmailSender fakeEmailSender; //تست
UserService service(fakeEmailSender);
service.Register();
assert(fakeEmailSender.called);
```

3- رعایت **SOLID** : به خصوص **Dependency Inversion Principle** که اصل پنجم SOLID است!

High-level modules should not depend on low-level modules.

در این رویکرد، ماژول‌های سطح بالا نباید به ماژول‌های سطح پایین وابسته باشند! باید هر دو به انتزاعات (Abstractions، مثل اینترفیس‌ها) وابسته باشند. انتزاعات نباید به جزئیات وابسته باشند، بلکه جزئیات باید به انتزاعات وابسته باشند.

در این مثال UserService نباید مستقیماً وابسته به EmailSender و Database و Logger باشد! بلکه باید به Abstraction وابسته شود!

 انواع **Dependency Injection** : سه روش اصلی وجود دارد.

1- **Constructor Injection**: که رایج‌ترین روش است! این روش بهترین روش در اکثر پروژه‌ها است!

```
class UserService
{
private:
    IEmailSender& _sender;

public:
    UserService(IEmailSender& sender)
        : _sender(sender)
    { }
};
```

مزایا: 

- ساده
- امن
- وابستگی‌ها اجباری هستند!

2- روش Setter Injection:

```
class UserService
{
private:
    IEmailSender* _sender;

public:
    void SetEmailSender(IEmailSender* s)
    {
        _sender = s;
    }
};
// استفاده
service.SetEmailSender(sender);
```

✓ مزایا:

- وابستگی اختیاری

✗ معایب:

- امکان فراموش کردن تنظیم وابستگی

3- روش Interface Injection:

این روش کم استفاده‌ترین روش در بین سه روش مذکور است! در این روش کلاس سطح

بالایی که باید یک یا چند شی را به شکل Interface تزریق شده دریافت کند، اینترفیسی را Implement می‌کند که او را مجبور می‌کند تابع یا توابع مختلف Inject یا تزریق را پیاده‌سازی کند تا اشیای مورد نیاز را در قالب انتزاعی دریافت کند!

```
class IInjectable
{
public:
    virtual void Inject(IEmailSender*) = 0;
};
```

🧠 مثال:

```
// اینترفیس تزریق توسط فریم‌ورک Container تعریف می‌شود
template<typename T>
class IInjectable {
public:
    virtual void Inject(T dependency) = 0;
    virtual ~IInjectable() = default;
};

// کلاس مصرف‌کننده که مجبور به پیاده‌سازی اینترفیس است
class OrderService : public IInjectable<std::shared_ptr<IPaymentGateway>> {
    std::shared_ptr<IPaymentGateway> gateway;
public:
    // این متد توسط کانتینر صدا زده می‌شود نه توسط ما
    void Inject(std::shared_ptr<IPaymentGateway> dependency) override {
        gateway = dependency;
    }
};
```

```

void ProcessOrder(double amount) {
    if (gateway) gateway->Pay(amount);
}

};

// نحوه کار Container
class DIContainer {
public:
    template<typename T>
    static void Resolve(T& client) {
        auto dep = std::make_shared<StripeGateway>();
        client.Inject(dep); // تزریق توسط کانتینر
    }
};

// استفاده
int main() {
    OrderService service;
    DIContainer::Resolve(service); // کانتینر خودش اینجکت را صدا می زند
    service.ProcessOrder(99.9);
}

```

در تزریق توسط Interface، کلاس مصرف‌کننده یک اینترفیس خاص را پیاده‌سازی می‌کند که متدی برای دریافت وابستگی دارد. این روش کمتر رایج است و پیچیدگی غیرضروری ایجاد می‌کند، مگر در فریم‌ورک‌های خاصی که این الگو را دیکته می‌کنند.

Dependency Injection Container چیست؟

در پروژه‌های بزرگ، ساخت دستی وابستگی‌ها سخت می‌شود.  مثال:

```

UserService
|
+--> UserRepository
    |
    +--> Database
        |
        +--> Config

```

 ساخت دستی:

```

Config config;
Database db(config);
UserRepository repo(db);
UserService service(repo);

```

در پروژه‌های بزرگ این کار پیچیده می‌شود و برای حل این مشکل از DI Container استفاده می‌کنیم.

DI Container چگونه کار می‌کند؟

ثبت وابستگی:

```

container.Register<IRepository, PostgresRepository>();

```

درخواست:

```

auto service = container.Resolve<UserService>();

```

Container خودش تمام وابستگی‌ها را می‌سازد. در واقع شما فقط یک بار به کانتینر می‌گویید: "هر وقت کسی `INotificationService` خواست، یک نمونه از `EmailSender` به او بده." سپس کانتینر به صورت بازگشتی (Recursive) تمام وابستگی‌ها را می‌سازد و تزریق می‌کند.

```
class Container {
private:
    std::unordered_map<std::type_index, std::function<void*>> factories;
public:
    template<typename Interface, typename Implementation>
    void Register()
    {
        factories[typeid(Interface)] =
            [] ()
            {
                return new Implementation();
            };
    }

    template<typename Interface>
    Interface* Resolve()
    {
        return static_cast<Interface*>(
            factories[typeid(Interface)] () );
    }
};
```

💡 نکته: void* یعنی اشاره‌گری از هر نوع داده! یعنی اشاره‌گری که می‌تواند از هر نوع نظیر int یا bool یا ClassA یا ClassB باشد!

🔑 یک مثال کاربردی از نحوه استفاده: فرض کنید یک Interface برای لاگر دارید و دو پیاده‌سازی مختلف:

```
class ILogger{
public:
    virtual void Log(const std::string& msg) = 0;
    virtual ~ILogger() = default;
};

class ConsoleLogger : public ILogger{
public:
    void Log(const std::string& msg) override{
        std::cout << "Console: " << msg << std::endl;
    }
};

class FileLogger : public ILogger {
public:
    void Log(const std::string& msg) override {
        //...نوشتن در فایل...
    }
};

/**حالا استفاده از Container شما**/
int main () {
    Container container;
    //ثبت نام: هر وقت ILogger خواستم، ConsoleLogger بساز
    container.Register<ILogger, ConsoleLogger>();
    //...
    //...
    //درخواست: به من یک ILogger بده (بدون اینکه بدانی دقیقاً چیست)
    ILogger* myLogger = container.Resolve<ILogger>();
    myLogger->Log("Hello World!");
    //خروجی: Console: Hello World!
```

```
delete myLogger; // فراموش نکنید که شی ساخته شده لاگر را حتما پاک کنید!
return 0;
}
```

نقد و بررسی: چرا این کد در C++ مدرن کمی قدیمی و خطرناک است؟

این کد از نظر منطقی کاملاً درست کار می‌کند، اما در ++C مدرن (C++11 به بعد) چند ایراد دارد:

1- **نشت حافظه (Memory Leak):** در Register از new استفاده شده، اما هیچ مکانیزمی برای delete کردن آن در Resolve وجود ندارد. کاربر نهایی باید بداند که باید delete کند!

2- **خطر static_cast روی void*:** اگر شما اشتباهاً container.Resolve<int> را صدا بزنید در حالی که int را Register نکرده‌اید، یا نوع اشتباهی را کست کنید، برنامه با رفتار تعریف‌نشده (Undefined Behavior) مواجه شده و کرش می‌کند.

نسخه مدرن و امن تر (پیشنهادی): در C++ مدرن، به جای void* و پوینترهای خام، از Smart Pointer ها استفاده می‌کنیم:

```
std::unordered_map<std::type_index, std::function<std::shared_ptr<void>()>>
factories;

template<typename Interface, typename Implementation>
void Register() {
    factories[typeid(Interface)] = []() {
        return std::make_shared<Implementation>();
    };
}

template<typename Interface>
std::shared_ptr<Interface> Resolve() {
    //shared_ptr<Interface> به shared_ptr<void> تبدیل
    return std::static_pointer_cast<Interface>(
        factories[typeid(Interface)]()
    );
}
```

در این نسخه مدرن، دیگر نیازی به delete کردن دستی نیست و مدیریت حافظه کاملاً خودکار است!

✂ خط دو مبحث Service Locator و DI : گاهی افراد این دو مورد را اشتباه می‌گیرند. در صورتیکه در ServiceLocator، خود کلاس وابستگی مورد نیازش را از یک کوله پشتی اشتراکی بین تمام اشیای پروژه پیدا می‌کند ولی در DI وابستگی از بیرون به کلاس تزریق می‌شود! (Service Locator در ادامه مباحث این کتاب توضیح داده خواهد شد!)

Service Locator:

```
auto logger = Locator::Get<ILogger>();
```

Dependency Injection:

```
UserService(ILogger& logger);
```

ارتباط DI با Design Patterns های دیگر: DI معمولاً با الگوهای زیر ترکیب می‌شود:

Factory , Repository , Strategy , Mediator , Observer , Unit Of Work

PresentationLayer → ApplicationLayer → DomainService → InfrastructureLayer

```
class CreateOrderHandler {
private:
    IOrderRepository& repository;
    IEventPublisher& publisher;
    IUnitOfWork& unitOfWork;
};
// Startup در
container.Register< IOrderRepository, PostgresOrderRepository>();

container.Register<IEventPublisher, KafkaPublisher>();

container.Register<IUnitOfWork, PostgresUnitOfWork>();
// سپس
auto handler = container.Resolve<CreateOrderHandler>();
```

✗ اشتباهات رایج در (سو) استفاده از Dependency Injection:

- 1- تزریق همه چیز: در معماری Domain Driven Design و معماری های زیرگروهش که در این کتاب به آنها پرداخته ایم، لایه ای به عنوان لایه مرکزی به عنوان لایه Domain یا Entity داریم! این Entity ها نباید وابستگی زیرساختی داشته باشند! این اشیا خاصیت کاملاً مستقل بدون نیازمندی به لایه های بیرونی دارند!

```
class User
{
public:
    User(ILogger& logger); // کلاس پوزر دامنه یا انتیتی است و وابسته به لاگر شده
};
```

- 2- وابستگی به Container : Container فقط در Composition Root استفاده می‌شود. Composition Root نقطه شروع برنامه اصلی شما یعنی تابع main() است و در همین نقطه تمام اشیای اصلی ساخته می‌شود و ما به ازای درخواست هر کلاس از اینترفیس مورد انتظار، نمونه Concrete و واقعی ساخته شده از آن اینترفیس را به وی می‌دهیم و لذا Container ای که شبه Map ای از اشیای مختلف عمل می‌کند را نباید به کلاس پاس دهیم تا آن کلاس هر آنچه لازم دارد را از آن بردارد! بلکه باید کلاس اصلی Concrete از اینترفیس مورد انتظار یک کلاس را به وی پاس دهیم!

```
class Service {
    Container container;
    // این کلاس، کانتینر اشیای اصلی را می‌گیرد
    // و هر آنچه که لازم دارد را از آن بر می‌دارد ولی این کار اشتباه است
};
```

- ترکیب (Composition Root): در نقطه ورود برنامه (مثلاً Program.cs یا یک IOC Container)، وابستگی‌ها به هم متصل می‌شوند: 

```
// انتخاب پیاده‌سازی مورد نظر
INotificationService myNotifier = new EmailSende();
OrderService orderService = new OrderService(myNotifier);
orderService.PlaceOrder();
```

تزریق بیش از حد: اگر Constructor شما این شکلی شد، احتمالاً کلاس چند مسئولیت دارد و اصل SRP (اصل اول Solid) نقض شده است!

```
Service (  
    A&,  
    B&,  
    C&,  
    D&,  
    E&,  
    F&,  
    G&) ;
```

بهترین شیوه‌ها (Best Practices)

✓ از Constructor Injection استفاده کنید.

✓ به Interface وابسته شوید. گاهی وقتها، بهتر است به **abstract** ها وابسته شوید! در جایی که مجبورید از **Diamond Inheritance** دوری کنید و از طرفی یک یا چند رفتار مشترک را در کلاسهای سطح بالاتر از نوع **Abstract** پیاده کنید تا مجبور نشوید در هر کلاس آنها را دوباره نویسی کنید!

✓ شی Container را فقط در Composition Root مثلاً تابع main نگه دارید.

✓ اشیای Domain Model را از DI Container مستقل نگه دارید.

✓ از DI برای تست‌پذیری استفاده کنید.

✓ اشیای Repository ها، Event Publisher ها، Command Bus ها و Unit Of Work ها را با DI تزریق کنید.

✿ تکمیل پازل دیدگاه ماژولار:

تصور کنید قصد دارید سیستمی طراحی کنید که هسته اصلی آن ثابت باشد، اما قابلیت‌های آن از طریق ماژول‌های خارجی (DLL ها) قابل گسترش یا تغییر باشند. در واقع می‌خواهیم نرم‌افزاری بسازیم که مثل یک لگو عمل کند؛ یعنی بتوانیم با اضافه یا کم کردن قطعات (DLL ها)، قابلیت‌های آن را تغییر دهیم.

در معماری سنتی (Tightly Coupled)، کلاس‌های سطح بالا (Core) مستقیماً به کلاس‌های سطح پایین (Implementation) وابسته‌اند. این یعنی هسته سیستم باید از قبل بداند که دقیقاً با چه کلاس‌هایی کار می‌کند و عملاً امکان تغییر یا افزودن ماژول جدید بدون تغییر کدهای هسته وجود ندارد. به بیان ساده تر، در روش قدیمی، کلاس اصلی ما مثل یک دستگاهی بود که قطعاتش به بدنه جوش داده شده بودند! اگر می‌خواستید قطعه‌ای را عوض کنید، باید کل دستگاه را می‌شکستید. کلاس سطح بالا مجبور بود تمام کلاس‌های سطح پایین را بشناسد و مستقیماً آن‌ها را بسازد (New کند). این یعنی پیش‌بینی تمام نیازهای آینده غیرممکن بود.

💡 راه‌حل با **Dependency Injection**: ما به جای وابستگی به "پیاده‌سازی‌ها"، به "قراردادها" (Interfaces) وابسته می‌شویم.

1- تعریف قرارداد: هسته اصلی سیستم فقط اینترفیس‌ها را می‌شناسد و نمی‌داند چه کسی قرار است آن‌ها را پیاده‌سازی کند. کلاس سطح بالا فقط آگهی می‌دهد و می‌گوید من می‌خواهم خدمت‌گذاری داشته باشم که ساعت فلان و فلان، خوردن قرص‌ها را یادم بیندازد، بدون اینکه بداند چه کسی این کار را انجام خواهد داد و کلاس خدمتگذار چه ویژگی‌های اختصاصی دارد و ...

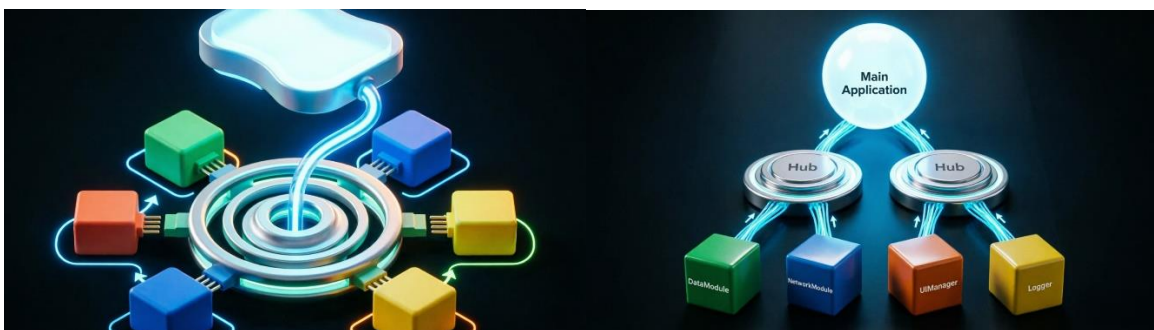
2- پیاده‌سازی خارجی: ماژول‌های جانبی (DLLها) اینترفیس‌ها را پیاده‌سازی (Implement) می‌کنند.

3- تزریق: در زمان اجرا، کاننیر DI، اشیای Concrete موجود در DLL ها را پیدا کرده و به عنوان وابستگی به کلاس‌های سطح بالا تزریق می‌کند.

به بیان ساده تر، کلاس سطح بالا فقط می‌گوید: «من به یک سرویس "پردازشگر" نیاز دارم که فلان کار را انجام دهد!» (بدون اینکه بداند این پردازشگر کیست)!

حالا هر DLL خارجی که بتواند این نقش را بازی کند (Interface را پیاده‌سازی کند)، می‌تواند به عنوان یک قطعه Plug-and-Play به سیستم وصل شود. به این ترتیب، ما لایه‌های تو در توری از DLL ها داریم که هر کدام وظیفه‌ای دارند، اما هسته اصلی نرم‌افزار هیچ‌کدام را مستقیماً نمی‌شناسد و فقط در لحظه آخر، قطعه مناسب را دریافت و استفاده می‌کند.

 **نتیجه:** شما یک معماری لایه‌ای و تودرتو دارید که در آن هسته مرکزی از جزئیات بی‌خبر است و تنها با قراردادهای کار می‌کند؛ این امر امکان افزودن، حذف یا جایگزین کردن DLLها و پلاگین‌ها را بدون نیاز به کامپایل مجدد هسته اصلی فراهم می‌سازد.



به این شکل از معماری پلاگین‌محور، معماری Plugin Architecture یا ماژولار کردن با وارونگی وابستگی می‌گویند.

 **تله استفاده سو از Interface به قصد DI:** فرض کنید می‌خواهید در Qt Framework با استفاده از الگوی MVC (در ادامه کتاب توضیح داده شده است)، یک نرم‌افزار تحت موبایل طراحی کنید! قرار است کلاسهای Model تمام صفحات از IModel به ارث بروند و صفحه Index اولیه هم پس از 5 ثانیه نمایش Logo شرکت به صفحه بعدی منتقل می‌شود و آنرا هم از IDurablePageModel پیاده‌سازی می‌کنید و IPageModelها هم به صورت کلی، آدرس صفحه دارند! ولی به جهت توهم پایبندی به DI استاندارد، تلاش می‌کنید به همه کلاسهایتان فقط و فقط Interface تزریق کنید و نه چیز دیگری! ساختار زیر را می‌چینید:

```
#include <QObject>
#include <QString>
// Base model: every model has a type.
class IModel : public QObject {
    Q_OBJECT
    Q_PROPERTY(QString type READ GetType)
public:
    using QObject::QObject;
    virtual QString GetType() const = 0;
};
// Page model: every page has an address.
class IPageModel : public IModel {
    Q_OBJECT
    Q_PROPERTY(QString pageAddress READ GetPageAddress)
```

```

public:
    using IModel::IModel;
    virtual QString GetPageAddress() const = 0;
};
// Durable page: a page with a configurable duration like index page which will move to
// the next page after a duration
class IDurablePageModel : public IPageModel {
    Q_OBJECT
    Q_PROPERTY(int duration READ GetDuration WRITE SetDuration)
public:
    using IPageModel::IPageModel;
    virtual int GetDuration() const = 0;
    virtual void SetDuration(int) = 0;
};

class PageModel: public IPageModel {
    Q_OBJECT
public:
    PageModel(const std::string& PageAddress):_page_address(PageAddress){}
    QString GetType() const override { return "PageModel"; }
    QString GetPageAddress() const override { return _page_address; }
private:
    std::string _page_address;
}
// Each page must implement GetPageAddress() separately.
Class IndexPageModel : public IdurablePageModel {
    Q_OBJECT
public:
    using IdurablePageModel::IdurablePageModel;
    QString GetType() const override { return "Index"; }
    QString GetPageAddress() const override { return "/index"; }
    int GetDuration() const override { return m_duration; }
    void SetDuration(int value) override { m_duration = value; }
private:
    int m_duration = 0;
};

class LoginPageModel : public IpageModel {
    Q_OBJECT
public:
    using IpageModel::IpageModel;
    QString GetType() const override { return "Login"; }
    QString GetPageAddress() const override { return "/login"; }
};

class SplashPageModel : public IpageModel {
    Q_OBJECT
public:
    using IpageModel::IModel;
    QString GetType() const override { return "Splash"; }
    QString GetPageAddress() const override { return "/splash"; }
};

```

همانطوری که در کد فوق مشاهده می کنید، به سادگی می توانید در **IPageModel** یک **Private Attribute** برای **PageAddress** و در **IDurablePageModel** یک **Private Attribute** برای **PageDuration** بسازید و **Getter** و **Setter** ها را هم در همان کلاسها تعریف کنید تا دیگر مجبور نباشید یک متد تکراری مثل ست کردن **PageAddress** را در کلاسهای زیرین تکرار کنید! ولی چون می خواستید به استانداردها پایبند باشید و **Interface** هایتان هیچ متدی را پیاده سازی نکنند، ترجیح دادید یک مسیر طولانی را حرکت کنید تا جایی که به دردسر تعریف متدهای تکراری بیفتید!

نکته بسیار مهم: اصل DI شما را مجبور به استفاده از Interface ها نمی کند، هر چند اگر ممکن باشد، بهتر است واسطه انتقال وابستگی Interface باشد! به بیان بهتر اگر در چنین سناریوهایی گرفتار شدید، می توانید به جای Interface از Abstract Class ها استفاده کنید! در زیر، کد اصلاح شده را می بینید!

```
#include <QObject>
#include <QString>
// Abstract base model.
class IModel : public QObject {
    Q_OBJECT
    Q_PROPERTY(QString type READ GetType)
public:
    using QObject::QObject;
    virtual QString GetType() const = 0;
};
// Abstract page model: GetPageAddress() is shared.
class IPageModel : public IModel {
    Q_OBJECT
    Q_PROPERTY(QString pageAddress READ GetPageAddress)
public:
    using IModel::IModel;
    QString GetPageAddress() const { return m_pageAddress; }

protected:
    explicit IPageModel(const QString& address, QObject* parent = nullptr)
        : IModel(parent), m_pageAddress(address) {}

private:
    QString m_pageAddress;
};
// Abstract durable page: duration behavior is shared.
class IDurablePageModel : public IPageModel {
    Q_OBJECT
    Q_PROPERTY(int duration READ GetDuration WRITE SetDuration)
public:
    using IPageModel::IPageModel;

    int GetDuration() const { return m_duration; }
    void SetDuration(int value) { m_duration = value; }

private:
    int m_duration = 0;
};
// Pages only implement what is actually different.
class IndexPageModel : public IDurablePageModel {
    Q_OBJECT
public:
    using IDurablePageModel::IDurablePageModel;
    QString GetType() const override { return "Index"; }
};

class LoginPageModel : public IPageModel {
    Q_OBJECT
public:
    using IPageModel::IPageModel;
    QString GetType() const override { return "Login"; }
};
```

```
class SplashPageModel : public IPageModel {
    Q_OBJECT
public:
    using IPageModel::IPageModel;
    QString GetType() const override { return "Splash"; }
};
```

پس با اصلاح رویکرد، می توانید از تعریف متدهای تکراری دوری کنید و رمز این کار، استفاده از **Abstract Class** ها است و فهم این موضوع که **DI** شما را مجبور نمی کند فقط از **Interface** ها استفاده کنید! بلکه قرار است مسیر وابستگی به جای وابستگی از کلاسهای استفاده کننده به سمت اصل کلاسهای خدمتکار، به سمت وابستگی کلاسهای خدمتکار به کلاس های استفاده کننده یا سرویس یا **Interface** مشترک بین آنها در **common.h** و ... تغییر یابد!

جمع بندی

الگوی تزریق وابستگی (**DI**) یک ترند کدنویسی نیست، بلکه یک تغییر پارادایم در نحوه تفکر درباره طراحی نرم افزار است. با جدا کردن "ایجاد" اشیاء از "استفاده" از آنها، **DI** به ما کمک می کند تا نرم افزاری بسازیم که:

- در برابر تغییرات مقاوم است.
- به راحتی قابل تست است.
- در طول زمان مقیاس پذیر و قابل نگهداری باقی می ماند.

اگر می خواهید کدی بنویسید که اصول **SOLID** را رعایت کند و برای محیط های حرفه ای و تیمی آماده باشد، تسلط بر الگوی **Dependency Injection** و استفاده از یک **IoC Container**، یک ضرورت مطلق است.

یادتان باشد که در **C++** می توانید ساختار و **Interface** یک کلاس را در فایل **h**. یا **hpp** قرار دهید و پیاده سازی آن را به جای فایل **cpp**، در یک **.dll**، **.so** یا سایر انواع **Dynamic Library** مدیریت کنید. درون این **Dynamic Library** نیز می توانید کلاس های داخلی یا **Local Class** تعریف کنید و حتی **Interface** آن ها را در همان **DLL** قرار داده و جزئیات پیاده سازی را در یک **Dynamic Library** دیگر مدیریت کنید.

به این ترتیب می توان یک سیستم کاملاً ماژولار ایجاد کرد؛ به گونه ای که با اضافه یا حذف کردن **DLL** های مرتبط، قابلیت های مختلف به نرم افزار اضافه یا از آن حذف شوند. در این مدل، نرم افزار استفاده کننده نیازی به شناخت جزئیات داخلی این **DLL** ها ندارد و می تواند قابلیت های جدیدی را از طریق آن ها دریافت و استفاده کند؛ حتی قابلیت هایی که هنگام طراحی اولیه نرم افزار پیش بینی نشده بودند.

 **تمرین:** فرض کنید می خواهید یک سیستم پرداخت بسازید که هسته اصلی آن در **DLL A** است، اما روش های پرداخت (کارت، کیف پول، کریپتو) در **DLL** های جداگانه پیاده سازی می شوند. این پروژه را با استفاده از **DI** طراحی کنید!

فصل دوم

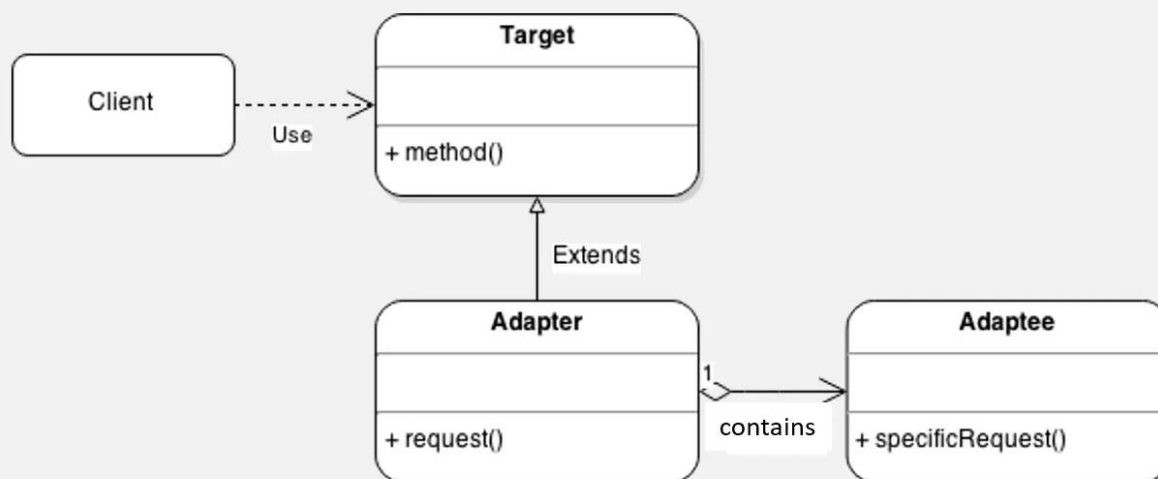
دیزاین پترنهای ساختاری (Structural)

این دسته از دیزاین پترن‌ها، به سلسله مراتب (Hierarchy) و نحوه ترکیب کلاس‌ها و اشیاء برای تشکیل ساختارهای بزرگتر متمرکز می‌شوند. دیزاین پترنهای ساختاری کلاسی، از ارث‌بری برای ترکیب رابط‌ها یا پیاده‌سازی‌ها استفاده می‌کنند. انواع دیزاین پترنهای ساختاری عبارتند از:

○ **Adapter**: پترنی از نوع Structural است!



Adapter pattern – Class diagram



در دنیای نرم‌افزار، اغلب با شرایطی روبرو می‌شویم که یک کلاس یا کامپوننت موجود، اینترفیسی دارد که با نیازهای جدید ما سازگار نیست. در چنین مواقعی، الگوی آداپتور (**Adapter Pattern**) یکی از بهترین راه‌حل‌هاست. هدف این الگو این است که دو اینترفیس ناسازگار را به گونه‌ای به هم متصل کند که بدون تغییر در کد منبع، بتوانند با یکدیگر کار کنند.

مثال کاربردی: درگاه‌های پرداخت بانکی

تصور کنید شما یک پلتفرم فروش آنلاین توسعه داده‌اید و می‌خواهید از چندین درگاه پرداخت مختلف مثل درگاه بانک ملی، سپرده و یا زرین‌پال پشتیبانی کنید. هر یک از این درگاه‌ها، API های متفاوتی دارند:

- بانک ملی ممکن است متد `processPayment(cardNumber, amount)` را ارائه دهد.
- زرین‌پال از `requestPayment(amount, mobile)` استفاده کند.
- سپرده هم شاید `executeTransaction(data)` را پیشنهاد دهد.

اما در لایه‌ی کسب‌وکار برنامه شما، دوست دارید یک اینترفیس یکسان داشته باشید، مثلاً:

```
virtual bool pay(double amount, const string& cardInfo) = 0;
```

در اینجا، کلاس‌های آداپتور وارد عمل می‌شوند. پس برای هر درگاه، یک آداپتور می‌سازیم که اینترفیس استاندارد شما را پیاده‌سازی کند و درون آن، به API خاص آن بانک متصل شود.

```
class MellatAdapter : public PaymentProcessor {
public:
    bool pay(double amount, const string& cardInfo) override {
        MellatBank bank;
        return bank.processPayment(cardInfo, amount); // تبدیل فراخوانی
    }
};

class ZarinpalAdapter : public PaymentProcessor {
public:
    bool pay(double amount, const string& cardInfo) override {
        ZarinpalService service;
        return service.requestPayment(amount, extractMobile(cardInfo));
    }
};
```

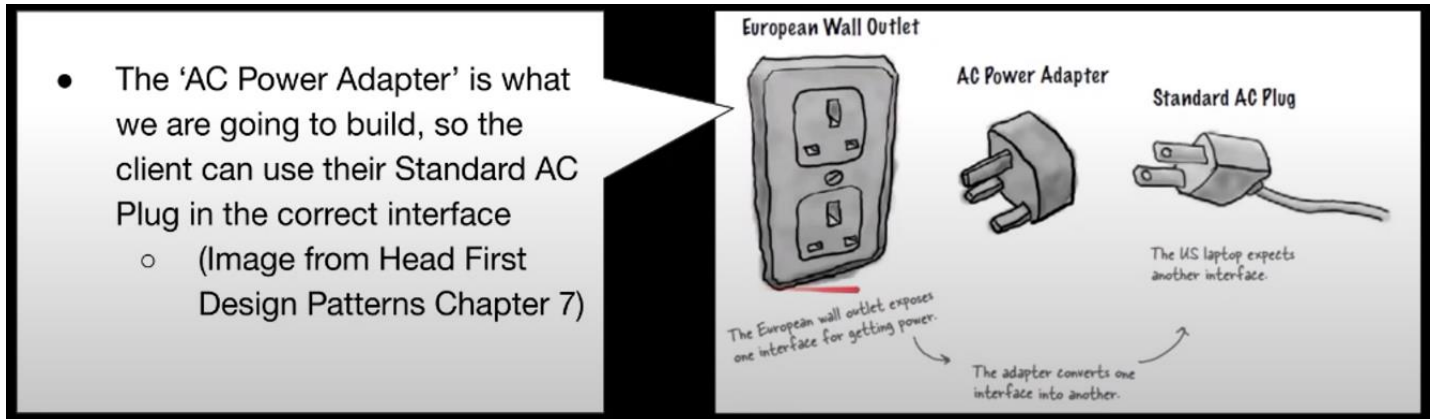
به این ترتیب، هسته اصلی برنامه شما فقط با `PaymentProcessor` کار می‌کند و نیازی به شناخت جزئیات هر درگاه ندارد. این همان هدف اصلی آداپتور است: هماهنگ‌سازی بدون تغییر در کد موجود.

🌸 دو نوع آداپتور داریم:

1. آداپتور شیء‌محور (**Object Adapter**): از کامپوزیشن استفاده می‌کند (مانند مثال بالا).
 2. آداپتور کلاس‌محور (**Class Adapter**): در زبان‌هایی مانند `C++` با ارث‌بری چندگانه پیاده‌سازی می‌شود (کمتر رایج و خطرناک‌تر).
- در عمل، آداپتور شیء‌محور انعطاف‌پذیرتر و ایمن‌تر است.

- اتصال کتابخانه‌های شخص ثالث قدیمی به سیستم‌های جدید
- ادغام سرویس‌های میکروسرویسی با پروتکل‌های متفاوت (REST vs gRPC)
- تطبیق داده‌های دریافتی از سنسورهای صنعتی با مدل داخلی سیستم

پس در حقیقت این دیزاین پترن تلاش دارد wrapper هایی بوجود آورد تا یک اینترفیس خاص را به اینترفیس خاص دیگری تبدیل نماید!



کار یک آداپتور برق، تبدیل نوع برق ورودی است. به بیان بهتر وقتی برق ورودی مناسب دستگاه نباشد، آداپتور تلاش می کند آنرا مناسب دستگاه تبدیل نماید! فرض کنید می خواهید آداپتوری برای `sf::Text` بنویسید:

```
// simple object adapter header
class AdapterText {
public:
    AdapterText(sf::Text text);
    void draw(sf::RenderWindow& window);
private:
    sf::Text m_internalText;
};

//simple object adapter cpp
AdapterText::AdapterText(sf::Text text){
    m_internalText = text;
    m_internalText.setCharacterSize(24);
    m_internalText.setFillColor(sf::Color::Green);
    std::string original = text.getString();
    for(size_t i=0; i < original.length();i++){
        original[i] = std::toupper(original[i]);
    }
    m_internalText.setString(original);
}

void AdapterText::draw(sf::RenderWindow& window){
    window.draw(m_internalText);
}

//use: create an adapter sometime after! Our previous text was created
AdapterText adapterText(text);
```

🎮 **مثالی دیگر از آداپتر:** فرض کنید شبی از کلاسی داریم که به زبان غیر قابل فهم شما پیام می دهد. مثلا پیامهایش را معکوس به شما می دهد. در مثال زیر قرار است آداپتری برایش بنویسیم تا پیام آنرا به صورت اصلاح شده به شما برگرداند:

```

//The Target defines the domain-specific interface used by the client code.
class Target {
public:
    virtual ~Target() = default;

    virtual std::string Request() const {
        return "Target: The default target's behavior.";
    }
};

// The Adaptee contains some useful behavior, but its interface is incompatible
// with the existing client code. The Adaptee needs some adaptation before the
// client code can use it.
class Adaptee {
public:
    std::string SpecificRequest() const {
        return ".eetpadA eht fo roivaheb laicepS";
    }
};

// The Adapter makes the Adaptee's interface compatible with the Target's
// interface.
class Adapter : public Target {
private:
    Adaptee *adaptee_;
public:
    Adapter(Adaptee *adaptee) : adaptee_(adaptee) {}
    std::string Request() const override {
        std::string to_reverse = this->adaptee_->SpecificRequest();
        std::reverse(to_reverse.begin(), to_reverse.end());
        return "Adapter: (TRANSLATED) " + to_reverse;
    }
};

// The client code supports all classes that follow the Target interface.
void ClientCode(const Target *target) {
    std::cout << target->Request();
}

int main() {
    std::cout << "Client: I can work just fine with the Target objects:\n";
    Target *target = new Target;
    ClientCode(target);
    std::cout << "\n\n";
    Adaptee *adaptee = new Adaptee;
    std::cout << "Client: The Adaptee class has a weird interface. See, I don't
understand it:\n";
    std::cout << "Adaptee: " << adaptee->SpecificRequest();
    std::cout << "\n\n";
    std::cout << "Client: But I can work with it via the Adapter:\n";
    Adapter *adapter = new Adapter(adaptee);
    ClientCode(adapter);
    std::cout << "\n";

    delete target;
    delete adaptee;
    delete adapter;

    return 0;
}

```

همان طوری که می بینید، کلاس آداپتر شامل اشاره گر یا داده ای از نوع نامانوس است و در تابع خاصش، آنرا تبدیل به مقداری می کند که به درد کلاس استفاده کننده از آداپتر می خورد.

مثالی دیگر که کمک می کند یک لاگر قدیمی را در دل یک لاگر جدید به عنوان آداپتر قرار دهیم تا از آن استفاده کنیم بدون اینکه در زمان اجرا از آن اطلاع داشته باشیم:

```
#include <iostream>
#include <memory>
// کلاس قدیمی با اینترفیس ناسازگار
class OldLogger {
public:
    void logMessage(const std::string& msg) {
        std::cout << "[Old] Logging: " << msg << std::endl;
    }
};
// اینترفیس جدیدی که سیستم انتظارش را دارد
class NewLogger {
public:
    virtual ~NewLogger() = default;
    virtual void write(const std::string& text) = 0;
};
// آداپتور: دو اشاره گر دارد - یکی به OldLogger و یکی به اینترفیس (NewLogger از طریق
class LoggerAdapter : public NewLogger {
private:
    std::unique_ptr<OldLogger> oldLogger; // اشاره گر به سرویس قدیمی
public:
    LoggerAdapter(std::unique_ptr<OldLogger> old) : oldLogger(std::move(old)) {}

    void write(const std::string& text) override {
        oldLogger->logMessage(text); // تطبیق فراخوانی
    }
};
// مصرف کننده سیستم جدید
void useNewLogger(NewLogger* logger) {
    logger->write("Hello from new system!");
}
// مثال اجرا
int main() {
    auto old = std::make_unique<OldLogger>();
    auto adapter = std::make_unique<LoggerAdapter>(std::move(old));
    useNewLogger(adapter.get()); // OldLogger از آگاهی بدون
    return 0;
}
```

🧠 در مثال بعدی آداپتر: از کامپوزیشن استفاده می کنیم (نه ارث بری از کلاس Client)؛ در این مثال آداپتور فقط به سرویس قدیمی (Adaptee) وابسته است! فضاپیما (Client) از وجود ربات قدیمی بی خبر است،

```
#include <iostream>
#include <memory>
#include <string>
#include <unordered_map>
// اینترفیس استاندارد که سیستم جدید (فضاپیما) انتظارش را دارد
class IRobot {
public:
    virtual ~IRobot() = default;
```

```

    // اجرای یک دستور به زبان استاندارد (مثلاً انگلیسی)
    virtual void execute(const std::string& command) = 0;
};

// سرویس قدیمی: ربات زمینی که فقط فارسی می‌فهمد
// (این کلاس را نمی‌توان تغییر داد - مثلاً کتابخانه شخص ثالث است)
class GroundRobot {
public:
    void executeFarsiCommand(const std::string& farsiCommand) {
        std::cout << "[GroundRobot] دستور : " << farsiCommand << '\n';
    }
};

// آداپتور: تطبیق ربات قدیمی با اینترفیس استاندارد
// فقط یک اشاره‌گر به ربات قدیمی دارد (کامپوزیشن) -
// فضاپیما را نمی‌شناسد و به آن وابسته نیست -
class RobotAdapter : public IRobot {
private:
    std::shared_ptr<GroundRobot> robot;
    std::unordered_map<std::string, std::string> commandMap = {
        {"move forward", "حرکت به جلو"},
        {"turn left", "چرخش به چپ"},
        {"stop", "توقف"},
        {"scan area", "اسکن محیط"}
    };
    // دیکشنری ساده برای ترجمه دستورات
public:
    explicit RobotAdapter(std::shared_ptr<GroundRobot> r)
        : robot(std::move(r)) {}
    void execute(const std::string& englishCommand) override {
        auto it = commandMap.find(englishCommand);
        if (it != commandMap.end()) {
            robot->executeFarsiCommand(it->second);
        } else {
            std::cout << "[RobotAdapter] دستور «" << englishCommand
                << "» قابل ترجمه نیست\n";
        }
    }
};

// سیستم جدید: فضاپیما
// فقط با IRobot کار می‌کند
// از وجود GroundRobot یا زبان فارسی هیچ خبری ندارد
class Spaceship {
private:
    std::shared_ptr<IRobot> robot; // تنها وابستگی: به اینترفیس ←
public:
    void setRobot(std::shared_ptr<IRobot> r) {
        robot = std::move(r);
    }
    void sendCommand(const std::string& cmd) {
        std::cout << "[Spaceship] sends English command: " << cmd << '\n';
        if (robot) {
            robot->execute(cmd); // فکر می‌کند ربات مستقیماً انگلیسی می‌فهمد
        }
    }
};

// مثال اجرا - ساختار تمیز و بدون جزئیات اضافه
int main() {
    // ساخت ربات قدیمی
    auto oldRobot = std::make_shared<GroundRobot>();
    // ساخت آداپتور که ربات قدیمی را "پوشش" می‌دهد
    auto adapter = std::make_shared<RobotAdapter>(oldRobot);
    Spaceship ship; // فضاپیما فقط با اینترفیس کار می‌کند
    ship.setRobot(adapter);
}

```



```
// ارسال دستورات - بدون نیاز به تغییر در کد فضاپیما
ship.sendCommand("move forward");
ship.sendCommand("turn left");
ship.sendCommand("hover"); // دستور ناشناخته
return 0;
}
```

خروجی این برنامه: 

```
[Spaceship] sends English command: move forward
```

```
[GroundRobot] اجرای دستور: حرکت به جلو
```

```
[Spaceship] sends English command: turn left
```

```
[GroundRobot] اجرای دستور: چرخش به چپ
```

```
[Spaceship] sends English command: hover
```

```
[RobotAdapter] قابل ترجمه نیست «hover» دستور
```

 در مثالی دیگر می‌خواهیم از یک کلاس جدید معمولاً Third Party استفاده کنیم، اما این کلاس با ساختار سیستم ما سازگار نیست.

در این مثال، الگوی Adapter کمک می‌کند:

- دو Interface ناسازگار را به هم وصل کنیم!
- بدون تغییر در کدهای Third Party آن را سازگار کنیم!
- سیستم ما یک API یکپارچه داشته باشد!

سناریوی مثال — فیلترهای تصویر

- یک کلاس Image داریم!
- یک کلاس ImageView داریم که روی تصویر فیلتر اعمال می‌کند!
- یک اینترفیس ساده به نام Filter داریم!
- چند فیلتر معمولی مثل VividFilter داریم!
- یک فیلتر خارج از سیستم ("third-party") داریم به نام Caramel!

 اما مشکل اصلی: کلاس Caramel در واقع API های خاص خود را دارد و به جای apply() از متدهای init() و render() استفاده می‌کند و طبیعتاً با اینترفیس Filter سازگار نیست.

ساختمان اولیه سیستم 

1- کلاس Image:

```
class Image {}
```

2- اینترفیس Filter :

```
class Filter {
public:
    virtual void apply(Image* image) = 0;
}
```

3- یک فیلتر معمولی **VividFilter**: این همان فیلتر "نرمال" ماست که از **interface** سیستم خودمان تبعیت می‌کند.

```
class VividFilter: public Filter {
public:
    virtual void apply(Image* image) {
        std::cout << "Applying Vivid filter";
    }
}
```

4- کلاس **ImageView**: این کلاس وظیفه دارد فیلتر انتخاب شده را روی تصویر اعمال کند.

```
class ImageView {
private:
    Image* image;
public:
    ImageView(Image* image):image(image) {}

    void apply(Filter* filter) {
        filter->apply(image);
    }
}
```

هر فیلتری که اینترفیس **Filter** را پیاده کند روی **ImageView** کار می‌کند.

💡 ورود کلاس **Third-Party – Caramel**: حالا یک کتابخانه خارجی داریم که داخل آن این کلاس وجود دارد:

```
class Caramel {
public:
    void init() {
        std::cout << "Initializing Caramel filter";
    }

    void render(Image& image) {
        std::cout << "Rendering image using Caramel filter";
    }
}
```

❌ مشکل چیست؟ این کلاس اصلاً اینترفیس **Filter** را پیاده نمی‌کند.

🔧 نیاز به **Adapter**: برای استفاده از **Caramel** داخل سیستم خود بدون دست‌زدن به کد کتابخانه **third-party** باید یک **Adapter** بسازیم!

📖 نوشتن **Adapter – CaramelFilterAdapter**

🔧 **Adapter** باید:

- اینترفیس **Filter** را پیاده‌سازی کند!

- از داخل خود یک نمونه Caramel نگه دارد!

- در متد apply()

- اول متد init() را اجرا کند

- سپس render() را فراخوانی کند

مثل زیر: 🍬

```
class CaramelFilterAdapter : public Filter {
private:
    Caramel* caramel;
public:
    CaramelFilterAdapter(Caramel* caramel) : caramel(caramel) {}
    void apply(Image* image) override {
        caramel->init();
        caramel->render(image);
    }
};
```

استفاده واقعی در کل سیستم: 🔑

```
#include <memory>
int main() {
    // ایجاد تصویر با make_unique
    auto image = std::make_unique<Image>();
    // ایجاد نما (View)
    // چون سازنده ImageView یک Image* می‌گیرد، از image.get() استفاده می‌کنیم
    auto view = std::make_unique<ImageView>(image.get());
    // اعمال فیلتر معمولی (VividFilter)
    // اگر متد apply فقط فیلتر را استفاده می‌کند و مالکیت آن را نمی‌گیرد: (Filter*)
    view->apply(std::make_unique<VividFilter>().get());
    // اعمال فیلتر Third Party با استفاده از Adapter
    // ابتدا Caramel را می‌سازیم
    auto caramel = std::make_unique<Caramel>();
    // سپس Adapter را با استفاده از اشاره‌گر Caramel ایجاد می‌کنیم
    auto adapter = std::make_unique<CaramelFilterAdapter>(caramel.get());
    // در نهایت Adapter را به apply پاس می‌دهیم
    view->apply(adapter.get());
    // نیازی به delete نیست، چون unique_ptr ها در پایان scope به صورت خودکار حافظه را آزاد می‌کنند
    return 0;
}
```

نتیجه خروجی: 🖨️

Applying Vivid filter

Initializing Caramel filter

Rendering image using Caramel filter

💡 چرا Adapter لازم است؟ ما نمی‌توانیم کدهای Caramel را تغییر بدهیم چون Third-Party است و تحت کنترل ما نیست.

همچنین نمی‌خواهیم سیستم ما پر از API‌های مختلف شود! اگر 10 کتابخانه مختلف بیاوریم و هر کدام متدهای مخصوص خود را داشته باشد، کد ما شلوغ و پیچیده می‌شود! Adapter باعث یکپارچگی سیستم می‌شود! ImageView فقط Filter می‌خواهد، مهم نیست چه فیلتری باشد! پس Adapter فقط واسطه است، نه تغییر دهنده سیستم داخلی! از طرفی ما با این روش، اصول SOLID مخصوصاً Interface Open-Closed و Segregation را رعایت می‌کنیم.

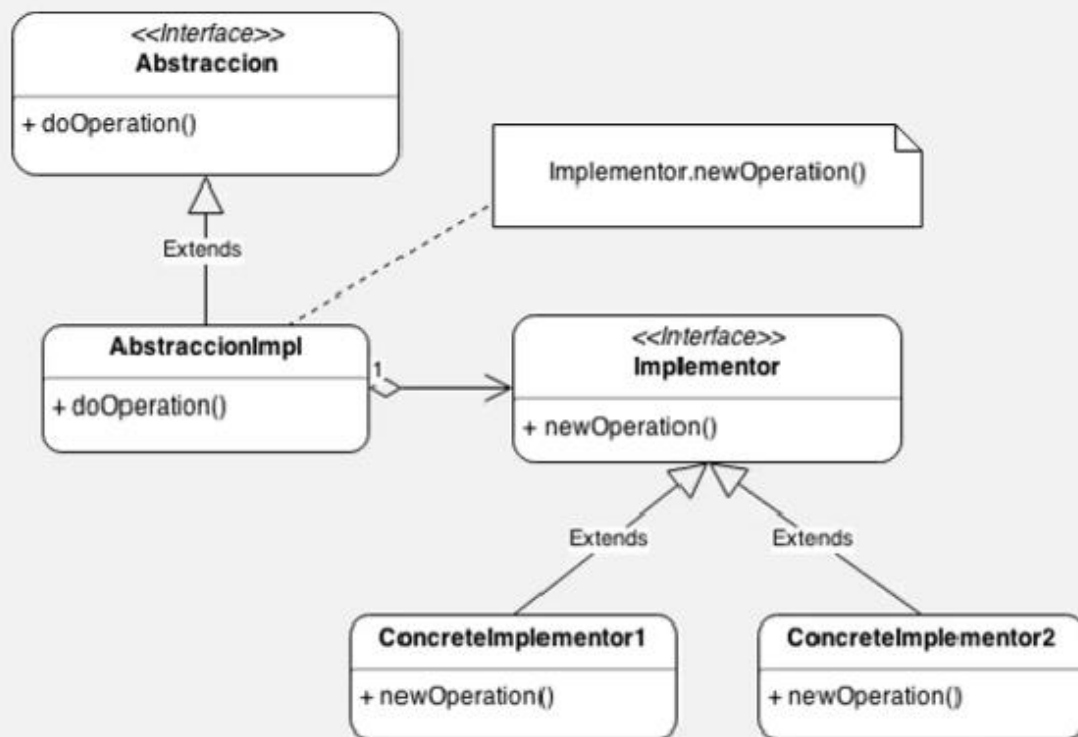
💡 نکته پایانی

الگوی آداپتور، الگویی ساده اما قدرتمند است که بدون ایجاد وابستگی مستقیم، امکان همکاری بین کامپوننت‌های ناسازگار را فراهم می‌کند و در معماری‌های مدرن مانند Clean Architecture و Hexagonal Architecture، از این الگو به‌طور گسترده برای جداسازی لایه‌ها و کاهش جفت‌شدگی (coupling) استفاده می‌شود.

Bridge: پترنی از نوع Structural است!



Bridge pattern – Class diagram



🏠 Bridge Pattern در C++ : جداسازی abstraction از implementation برای انعطاف پذیری بیشتر

در طراحی نرم افزار، یکی از اهداف اصلی، کاهش coupling بین بخش های مختلف سیستم است. گاهی یک کلاس همزمان دو یا چند بُعد از تغییر را در خود دارد. مثلاً ممکن است بخواهیم هم نوع Abstract (مثل یک کنترل گرافیکی Button, TextBox و ...) و هم نحوه ی پیاده سازی آن (مثل تم Light یا Dark) بتوانند به صورت مستقل تغییر کنند.

در چنین مواردی، استفاده از (Bridge Pattern الگوی پل) بسیار مفید است. این پترن به ما اجازه می دهد که abstraction (بخش انتزاعی یا رابط کاربری) را از implementation (پیاده سازی واقعی یا جزئیات) جدا کنیم، تا هر کدام بتوانند به صورت مستقل توسعه و تغییر پیدا کنند.

🏠 ساختار کلی Bridge Pattern: در این الگو معمولاً چهار بخش اصلی داریم:

1. Abstraction – رابط یا کلاس اصلی که کاربر با آن سر و کار دارد.
2. RefinedAbstraction – نسخه ی خاص تر از abstraction
3. Implementor – رابطی برای تعریف عملگرهایی که abstraction از آن ها استفاده می کند.
4. ConcreteImplementor – پیاده سازی واقعی رابط Implementor

ارتباط بین Abstraction و Implementor از طریق یک pointer یا reference است، نه ارث بری. این یعنی دو سلسله مراتب مستقل اما مرتبط داریم.

🎨 مثال: تنظیم تم در اپلیکیشن (Light/Dark Theme)

در این مثال، فرض کنید می خواهیم یک سیستم تم داشته باشیم که رنگ پس زمینه (background)، رنگ متن (foreground)، و radius (گردی گوشه ها) را برای کنترل های مختلف تنظیم کند. می خواهیم هم بتوانیم تم ها را تغییر دهیم (Dark / Light) و هم بتوانیم کنترل های مختلفی داشته باشیم مثلاً Button ، TextBox ، Label و

```
//Interface: ThemeImplementor.h
#pragma once
#include <string>
#include <memory>
#include <iostream>
class ThemeImplementor {
public:
    virtual ~ThemeImplementor() = default;
    virtual std::string backgroundColor() const = 0;
    virtual std::string foregroundColor() const = 0;
    virtual int borderRadius() const = 0;
};

//Concrete Implementations: DarkTheme / LightTheme
class DarkTheme : public ThemeImplementor {
public:
    std::string backgroundColor() const override { return "#222222"; }
    std::string foregroundColor() const override { return "#FFFFFF"; }
    int borderRadius() const override { return 8; }
};
```

```

class LightTheme : public ThemeImplementor {
public:
    std::string backgroundColor() const override { return "#FFFFFF"; }
    std::string foregroundColor() const override { return "#000000"; }
    int borderRadius() const override { return 4; }
};
//Abstraction: UIComponent
class UIComponent {
protected:
    std::shared_ptr<ThemeImplementor> theme;
public:
    UIComponent(std::shared_ptr<ThemeImplementor> impl)
        : theme(std::move(impl)) {}
    virtual ~UIComponent() = default;
    virtual void draw() const = 0;
};

class Button : public UIComponent { //Refined Abstractions: Button و Label
public:
    using UIComponent::UIComponent;
    void draw() const override {
        std::cout << "[Button] bg=" << theme->backgroundColor()
                    << ", fg=" << theme->foregroundColor()
                    << ", radius=" << theme->borderRadius() << "\n";
    }
};

class Label : public UIComponent {
public:
    using UIComponent::UIComponent;
    void draw() const override {
        std::cout << "[Label] bg=" << theme->backgroundColor()
                    << ", fg=" << theme->foregroundColor() << "\n";
    }
};
// استفاده در مین
int main() {
    std::shared_ptr<ThemeImplementor> dark = std::make_shared<DarkTheme>();
    std::shared_ptr<ThemeImplementor> light = std::make_shared<LightTheme>();

    Button darkButton(dark);
    Label darkLabel(dark);

    Button lightButton(light);
    Label lightLabel(light);

    darkButton.draw();
    darkLabel.draw();

    lightButton.draw();
    lightLabel.draw();

    return 0;
}

```

```
//خروجی
[Button] bg=#222222, fg=#FFFFFF, radius=8

[Label] bg=#222222, fg=#FFFFFF

[Button] bg=#FFFFFF, fg=#000000, radius=4

[Label] bg=#FFFFFF, fg=#000000
```

 **تحلیل مثال:** در اینجا کلاس‌های Button و Label از نوع Abstraction هستند و کلاس ThemelImplementor نقش Bridge Interface را دارد و کلاس‌های DarkTheme و LightTheme نقش Concrete Implementor را دارند.

مزیت مهم این طراحی این است که اگر بخواهیم تم جدیدی اضافه کنیم (مثلاً High Contrast Theme) نیازی به تغییر در کد Button و Label نداریم؛ کافی است یک کلاس جدید از ThemelImplementor بسازیم. همچنین می‌توانیم کنترل‌های جدیدی مثل TextBox یا Slider اضافه کنیم بدون اینکه تم‌ها را دست بزنیم.

در چه مواردی از Bridge Pattern استفاده می‌کنیم؟

الگوی Bridge زمانی مناسب است که دو بعد تغییر مستقل داریم (مثلاً نوع کنترل و نوع تم) و می‌خواهیم از وراثت بیش‌ازحد جلوگیری کنیم و نیاز داریم implementation را در زمان اجرا تغییر دهیم!

چند مثال دیگر:

- رندر گرافیکی: کلاس‌های مختلفی برای شکل‌ها (Circle, Rectangle) و پیاده‌سازی‌های متفاوت برای موتور گرافیکی مختلف و مدرن و امروزی مثل (OpenGL, DirectX, Vulkan)
- سیستم ذخیره‌سازی داده: برای DataStore و implementation های مختلف برای فایل، دیتابیس یا حافظه‌ی ابری.
- دستگاه‌های I/O: برای Device و implementation های مختلف برای USB، Bluetooth، یا Serial
- سیستم گزارش‌گیری (Logger) برای Logger و implementation برای ConsoleLogger، FileLogger یا RemoteLogger
- چند زبانه کردن نرم افزار با کمک رشته کلید انگلیسی مپ شده روی رشته ترجمه شده به زبانی که توسط Implementor لود شده!

 در اینجا، مورد آخر یعنی چند زبانه کردن نرم افزار توسط الگوی Bridge را مثال می‌زنیم!

```
#include <iostream>
#include <string>
#include <memory>
#include <unordered_map>
// 1. Implementor Interface (پیاده‌ساز)
// این کلاس فقط می‌داند چگونه یک رشته خام را بر اساس کلید پیدا کند
class LanguagePack {
public:
    virtual ~LanguagePack() = default;
```



```

    virtual std::string getRawString(const std::string& key) const = 0;
};
// 2. Concrete Implementors (پیاده‌سازهای ملموس)
class EnglishPack : public LanguagePack {
private:
    std::unordered_map<std::string, std::string> data = {
        {"btn.save", "Save"},
        {"btn.cancel", "Cancel"},
        {"msg.welcome", "Welcome to the app!"}
    };
public:
    std::string getRawString(const std::string& key) const override {
        auto it = data.find(key);
        return (it != data.end()) ? it->second : "";
    }
};
class PersianPack : public LanguagePack {
private:
    std::unordered_map<std::string, std::string> data = {
        {"btn.save", "ذخیره"},
        {"btn.cancel", "لغو"},
        {"msg.welcome", "به برنامه خوش آمدید"}
    };
public:
    std::string getRawString(const std::string& key) const override {
        auto it = data.find(key);
        return (it != data.end()) ? it->second : "";
    }
};
// 3. Abstraction (انتزاع)
// (Bridge) متصل است و به مدیریت می‌کند و این کلاس منطق ترجمه را
class Translator {
private:
    std::unique_ptr<LanguagePack> currentPack;
    std::unique_ptr<LanguagePack> fallbackPack; // زبان پیشفرض اگر کلید پیدا نشد
public:
    Translator(std::unique_ptr<LanguagePack> pack,
               std::unique_ptr<LanguagePack> fallback = nullptr)
        : currentPack(std::move(pack)), fallbackPack(std::move(fallback)) {}
    // (Runtime) متد برای تغییر زبان در زمان اجرا
    void setLanguage(std::unique_ptr<LanguagePack> pack) {
        currentPack = std::move(pack);
    }
    // متد اصلی که کلاینت با آن کار می‌کند
    std::string translate(const std::string& key) const {
        // 1. تلاش برای پیدا کردن در زبان فعلی
        std::string result = currentPack->getRawString(key);
        // 2. Fallback منطق (اگر در زبان فعلی نبود، زبان پیشفرض را چک کن)
        if (result.empty() && fallbackPack) {
            result = fallbackPack->getRawString(key);
        }
        // 3. اگر کلاً پیدا نشد، خود کلید را برگردان
        if (result.empty()) {
            return "[" + key + "]";
        }
        // اینجا می‌توانید منطق‌های پیچیده‌تری مثل جایگذاری متغیرها (Format) را اضافه کنید
        return result;
    }
};

```

```
// 4. Client Code (کد کلاینت)
int main() {
    auto englishPack = std::make_unique<EnglishPack>(); // ایجاد پک‌های زبانی
    auto persianPack = std::make_unique<PersianPack>();
    // ایجاد مترجم انگلیسی زبان پیش‌فرض یا Fallback است
    Translator translator(std::move(persianPack), std::make_unique<EnglishPack>());
    std::cout << "--- زبان فعلی: فارسی ---" << std::endl;
    std::cout << "Save Button: " << translator.translate("btn.save") << std::endl;
    std::cout << "Cancel Button: " << translator.translate("btn.cancel") << std::endl;
    // (Switching Language at Runtime) تغییر زبان در زمان اجرا
    std::cout << "\n--- تغییر زبان به انگلیسی ---" << std::endl;
    translator.setLanguage(std::make_unique<EnglishPack>());
    std::cout << "Save Button: " << translator.translate("btn.save") << std::endl;
    // تست Fallback: فرض کنید این کلید در پک انگلیسی نیست اما در فارسی هست
    // در این حالت چون زبان فعلی انگلیسی است و کلید را ندارد، به فارسی (Fallback) برمی‌گردد
    return 0;
}
```

نکته مهم: مثالی که در این بخش ارائه شد، صرفاً برای سادگی، رشته‌های مربوط به هر زبان را به صورت **Hard Code** در برنامه تعریف و بارگذاری می‌کند. در یک سیستم واقعی، معمولاً بهتر است فایل‌های زبان به صورت جداگانه و در قالب‌هایی مانند **XML** یا **JSON** در کنار نرم‌افزار قرار گیرند. (مثلاً **fa.json** و **en.json** و ...) سپس کلاس **Language Manager**، بر اساس زبان انتخاب‌شده توسط کاربر، فایل مربوطه را بارگذاری کرده و رشته‌های ترجمه‌شده را در **Data Table** مخصوص خود قرار می‌دهد.

📌 **نتیجه‌گیری:** **Bridge Pattern** یک الگوی قدرتمند در طراحی شیء‌گراست که انعطاف در توسعه و نگهداری کد را بسیار افزایش می‌دهد و با جدا کردن **abstraction** از **implementation**، می‌توان سیستم‌هایی ساخت که به راحتی گسترش پیدا کنند بدون آن‌که تغییرات یک بخش، موجب تغییر در بخش دیگر شود.

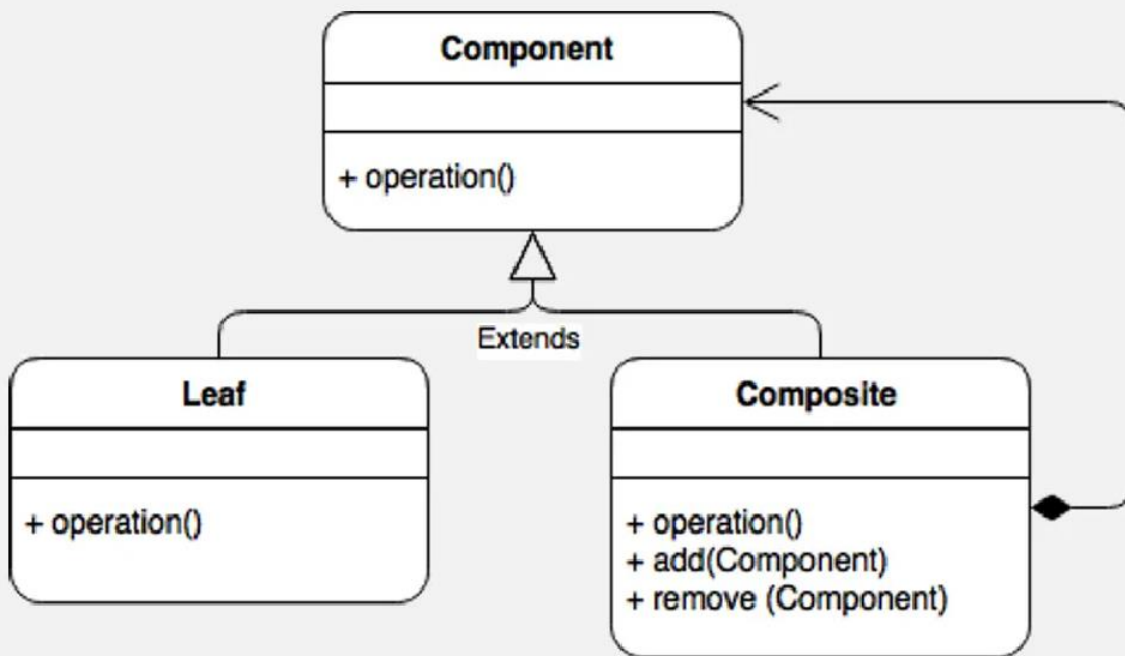
در مثال تم اپلیکیشن دیدیم که چگونه می‌توان با همین الگو، یک سیستم تم ساده اما انعطاف‌پذیر برای کنترل‌های گرافیکی طراحی کرد.

الگوی **Bridge** در طراحی ریموت کنترل تلویزیون باعث می‌شود رابط کنترل (مثل روشن/خاموش، تغییر کانال و صدا) از پیاده‌سازی تلویزیون جدا شود. به این ترتیب، یک نوع ریموت می‌تواند با برندها و مدل‌های مختلف تلویزیون کار کند، بدون اینکه برای هر ترکیب، کلاس ریموت جدیدی ایجاد کنیم. یعنی به جای اتصال مستقیم **RemoteControl** به **SamsungTV** یا **LGTV**، ریموت به یک **abstraction** مانند **TV** متصل می‌شود و پیاده‌سازی تلویزیون قابل تعویض است. به بیان بهتر همچنان که در مثال تغییر تم، **Implementor**های **Dark** و **Light** یک ادبیات عمومی و مشترک که رنگ پس زمینه و ... باشد را ارائه می‌دادند و هر بخش سیستم بر اساس مفهوم عمومی، تفسیر و عملکرد خاص خود را انتخاب می‌کردند و انجام می‌دادند، در ریموت کنترل هم ما با دستورات عمومی مشترک مثل روشن/خاموش شدن، کم کردن و زیاد کردن صدا، کانال بعدی/قبلی و ... روبرو هستیم و هر ریموت، دستور عمومی را فریاد می‌زد و در طرف مقابل، تلویزیون مدنظر دستور عمومی را دریافت می‌کند و بر اساس نوع و مدل خود، تفسیر دستور کاربر توسط ریموت کنترل را به سبک خود انجام می‌دهد و نتیجه می‌شود اجرای دستور عمومی کاربر به سبک صحیح خود! در مثال تغییر زبان نرم‌افزار هم برنامه نویس، ترجمه زبان مقصد را با کلید عمومی مشترک صدا می‌زد و **Language Implementor**ها هم بر اساس نوع زبان منتخب، رشته کلید را به زبان مقصد ترجمه می‌کردند و **return** می‌کردند!

Composite: پترنی از نوع Structural است!



Composite pattern – Class diagram



این الگو برای مدیریت ساختارهای درختی به شکل یکپارچه استفاده می شود!

در بسیاری از سیستم‌های نرم‌افزاری، با ساختارهای سلسله‌مراتبی (درختی) روبرو هستیم! فایل‌ها و پوشه‌ها در سیستم فایل، عناصر رابط کاربری یا UI، اجزای سخت‌افزاری، منوهای برنامه و غیره. در چنین ساختارهایی، هم گره‌های ساده (مثل یک دکمه یا یک فایل) و هم گره‌های مرکب (مثل یک

پنجره یا یک پوشه) وجود دارند. برای مثال شما می توانید دکمه و اسکرول بار و ... را داخل یک Page قرار دهید و با جا به جایی Page، آنها هم جابه جا می شوند! الگوی کامپوزیت (Composite Pattern) به ما امکان می دهد تا اجزای ساده و مرکب را به یک شکل یکسان مدیریت کنیم، گویی همه آنها یک نوع هستند. این کار با ایجاد یک اینترفیس مشترک برای هر دو نوع شیء (ساده و مرکب) انجام می شود.

🏠 ساختار کلی الگوی کامپوزیت

الگوی کامپوزیت از سه بخش اصلی تشکیل شده است :

Component	اینترفیس یا کلاس پایه ای که عملیات مشترک را تعریف می کند مثل Render یا GetPrice
Leaf	نماینده اجزای ساده است که هیچ زیرمجموعه ای ندارند مثل File یا Button
Composite	نماینده اجزای مرکب است که مجموعه ای از فرزندان (Child) دارد و معمولاً از نوع Component هستند مثل Folder یا Page یا Container

🌿 ویژگی کلیدی: کلاینت بدون نیاز به تمایز بین Leaf و Composite، با همه اجزا از طریق Component ارتباط برقرار می کند.

🎨 مثال ۱: سیستم رابط کاربری (UI System)

در یک UI Toolkit، ممکن است یک پنجره شامل پنل ها باشد، هر پنل شامل دکمه ها، برچسب ها و حتی پنل های داخلی دیگری باشد. این یک ساختار سلسله مراتبی کلاسیک است .

```
#include <iostream>
#include <vector>
#include <memory>
#include <string>
#include <algorithm> //std::find_if از استفاده
// 1. Component (کامپوننت پایه)
class UIElement {
protected:
    // انتقال نام به کلاس پایه برای جلوگیری از تکرار کد در کلاس های مشتق شده
    std::string name;
public:
    explicit UIElement(std::string n) : name(std::move(n)) {}
    virtual ~UIElement() = default;
    // متدهای اصلی
    virtual void render() const = 0;
    virtual std::string getName() const { return name; }
    // متدهای مدیریت ساختار (پیاده سازی پیش فرض برای Leaf ها)
    // بازگشت bool برای اطلاع از موفقیت یا شکست عملیات (ایمن تر از void)
    virtual bool addChild(std::unique_ptr<UIElement>) { return false; }
    virtual bool removeChild(const std::string&) { return false; }
    // بازگشت اشاره گر خام (Raw Pointer) زیرا ما مالک شیء نیستیم، فقط مشاهده گر آنیم
    virtual UIElement* getChild(const std::string&) { return nullptr; }
};
// 2. Leaf (ساده / عنصر برگ)
class Button : public UIElement {
public:
    // ارث بری سازنده از کلاس پایه برای جلوگیری از نوشتن کد تکراری
    using UIElement::UIElement;
```

```

    void render() const override {
        std::cout << "[Button] Rendering: " << name << '\n';
    }
};

class Label : public UIElement {
public:
    using UIElement::UIElement;

    void render() const override {
        std::cout << "[Label] Rendering: " << name << '\n';
    }
};

// 3. Composite (عناصر مرکب / کانتینر)
class Panel : public UIElement {
private:
    std::vector<std::unique_ptr<UIElement>> children;
public:
    using UIElement::UIElement;

    void render() const override {
        std::cout << "[Panel] Rendering: " << name
            << " (Contains " << children.size() << " elements)\n";
        // فراخوانی بازگشتی متد render برای تمام فرزندان
        for (const auto& child : children) {
            child->render();
        }
    }

    bool addChild(std::unique_ptr<UIElement> child) override {
        if (!child) return false; // بررسی اعتبار اشاره گر
        children.push_back(std::move(child));
        return true;
    }

    bool removeChild(const std::string& childName) override {
        // پیدا کردن اولین فرزند با نام مشخص
        auto it = std::find_if(children.begin(), children.end(),
            [&childName](const std::unique_ptr<UIElement>& child) {
                return child->getName() == childName;
            });
        // در صورت یافتن، از لیست حذف می شود و unique_ptr به صورت خودکار حافظه را آزاد می کند
        if (it != children.end()) {
            children.erase(it);
            return true;
        }
        return false;
    }

    UIElement* getChild(const std::string& childName) override {
        auto it = std::find_if(children.begin(), children.end(),
            [&childName](const std::unique_ptr<UIElement>& child) {
                return child->getName() == childName;
            });
        // برگرداندن اشاره گر خام (Non-owning pointer)
        return (it != children.end()) ? it->get() : nullptr;
    }
};

// 4. Client Code (نحوه استفاده)
int main() {
    // ایجاد عناصر مرکب و ساده
    auto mainWindow = std::make_unique<Panel>("Main Window");
    auto headerPanel = std::make_unique<Panel>("Header Panel");
    auto loginBtn = std::make_unique<Button>("Login Button");

```

```

auto titleLabel = std::make_unique<Label>("Title Label");
// ساخت درخت UI
mainWindow->addChild(std::move(headerPanel));
mainWindow->addChild(std::move(loginBtn));
// توجه: headerPanel توسط move مالکیت ندارد، اما می‌توانیم از طریق getChild به آن دسترسی داشته باشیم
// در اینجا فرض می‌کنیم یک Label دیگر به header اضافه می‌کنیم و نیاز به دسترسی به header داریم
// برای سادگی در این مثال، مستقیماً به mainWindow اضافه می‌کنیم
mainWindow->addChild(std::make_unique<Label>("Status Label"));
std::cout << "--- Initial Render ---\n";
mainWindow->render();
// استفاده از getChild
std::cout << "\n--- Finding Child ---\n";
UIElement* foundChild = mainWindow->getChild("Login Button");

if (foundChild) {
    std::cout << "Found: " << foundChild->getName() << '\n';
}
// استفاده از removeChild
std::cout << "\n--- Removing Child ---\n";

if (mainWindow->removeChild("Login Button")) {
    std::cout << "Login Button removed successfully.\n";
}
std::cout << "\n--- Final Render ---\n";
mainWindow->render();
return 0;
}

```

نتیجه خروجی: 

```

--- Initial Render ---
[Panel] Rendering: Main Window (Contains 3 elements)
[Panel] Rendering: Header Panel (Contains 0 elements)
[Button] Rendering: Login Button
[Label] Rendering: Status Label
--- Finding Child ---
Found: Login Button
--- Removing Child ---
.Login Button removed successfully
--- Final Render ---
[Panel] Rendering: Main Window (Contains 2 elements)
[Panel] Rendering: Header Panel (Contains 0 elements)
[Label] Rendering: Status Label

```

💡 نکته کلیدی: تابع render() برای همه اجزا یکسان است! چه ساده باشند، چه مرکب!

🧠 مثال ۲: اجزای کامپیوتر (Computer Components)

یک کامپیوتر از اجزای مختلفی تشکیل شده CPU، RAM، Hard و همچنین ماژول‌های مرکب مثل سیستم ذخیره‌سازی که ممکن است شامل SSD و HDD باشد.

```
#include <iostream>
#include <vector>
#include <memory>
#include <string>

class HardwareComponent { // Component
public:
    virtual ~HardwareComponent() = default;
    virtual void describe() const = 0;
    virtual double getPrice() const = 0;
    virtual void add(std::unique_ptr<HardwareComponent>) {} // فقط برای کامپوزیت
};

class CPU : public HardwareComponent { // Leaf
public:
    void describe() const override { std::cout << "- CPU (Intel i7)\n"; }
    double getPrice() const override { return 300.0; }
};

class RAM : public HardwareComponent {
public:
    void describe() const override { std::cout << "- RAM (16GB DDR4)\n"; }
    double getPrice() const override { return 80.0; }
};

class StorageSystem : public HardwareComponent { // Composite
private:
    std::vector<std::unique_ptr<HardwareComponent>> parts;
public:
    void describe() const override {
        std::cout << "Storage System:\n";
        for (const auto& part : parts) {
            part->describe();
        }
    }
    double getPrice() const override {
        double total = 0;
        for (const auto& part : parts) {
            total += part->getPrice();
        }
        return total;
    }
    void add(std::unique_ptr<HardwareComponent> part) override {
        parts.push_back(std::move(part));
    }
};

class Computer : public HardwareComponent { // کامپوزیت سطح بالاتر، کل کامپیوتر
private:
    std::vector<std::unique_ptr<HardwareComponent>> components;
public:
    void describe() const override {
        std::cout << "=== Computer Configuration ===\n";
        for (const auto& comp : components) {
            comp->describe();
        }
    }
}
```

```

double getPrice() const override {
    double total = 0;
    for (const auto& comp : components) {
        total += comp->getPrice();
    }
    return total;
}
void add(std::unique_ptr<HardwareComponent> comp) override {
    components.push_back(std::move(comp));
}
};

```

نحوه استفاده: 

```

int main() {

    auto computer = std::make_unique<Computer>();

    computer->add(std::make_unique<CPU>());

    computer->add(std::make_unique<RAM>());

    auto storage = std::make_unique<StorageSystem>();

    storage->add(std::make_unique<RAM>()); // به عنوان کش ذخیره سازی RAM فرض کنیم

    computer->add(std::move(storage));

    computer->describe();

    std::cout << "Total Price: $" << computer->getPrice() << '\n';

    return 0;

}

```

خروجی: 

```

=== Computer Configuration ===

- CPU (Intel i7)

- RAM (16GB DDR4)

Storage System:

- RAM (16GB DDR4)

Total Price: $460

```

نکته کلیدی:  محاسبه قیمت یا توصیف ساختار، به طور بازگشتی روی تمام زیرمجموعه‌ها اعمال می‌شود، بدون نیاز به تشخیص نوع شیء!

✓ مزایای الگوی کامپوزیت

- یکپارچگی رفتار: کلاینت با همه اجزا به یک شکل کار می‌کند.
- گسترش‌پذیری آسان: افزودن اجزای جدید (ساده یا مرکب) بدون تغییر در کد کلاینت.
- پشتیبانی از ساختارهای تو در تو: هر سطح از درخت می‌تواند به همان شکل مدیریت شود.

✗ معایب و چالش‌ها

- ایمنی نوع (Type Safety): در سطح اینترفیس، ممکن است کلاینت متدی مثل addChild() را روی یک Leaf فراخوانی کند که در Child نباشد یا رفتار صحیحی را انجام ندهد (البته در C++/Java معمولاً این متد در Leaf خالی است!)
- پیچیدگی طراحی: اگر تفاوت زیادی بین عملیات همانام واقع در Leaf و Composite وجود داشته باشد، ممکن است نیاز به طراحی جداگانه باشد (در این صورت، الگوی Visitor کمک می‌کند!)

چه زمانی از کامپوزیت استفاده کنیم؟

- وقتی سیستم شما ساختار درختی (سلسله مراتبی) دارد.
- وقتی می‌خواهید اجزای ساده و مرکب را یکسان ببینید.
- وقتی عملیات بازگشتی (مثل render، calculate، traverse) نیاز است.
- به صورت کلی بیشترین کارایی آنرا در ساختار فایل و فولدر، در UI با داشتن عناصری مانند Button و Label که داخل Page یا Container قرار می‌گیرند، در ساختار استخوان بندی عناصر 3D مثل کاراکترهای انسان در بازیها که بخش های مختلف همچون سر و بازوها و پاها به واسطه مفاصل به یکدیگر و بدن متصل می شوند و ... از این ساختار استفاده می شود!

📌 جمع‌بندی:

الگوی کامپوزیت یکی از زیباترین و کاربردی‌ترین الگوهای ساختاری است که به ما کمک می‌کند تا جهان پیچیده سلسله‌مراتب‌ها را به سادگی مدیریت کنیم. چه در طراحی رابط‌های کاربری، چه در مدل‌سازی سخت‌افزار، چه در سیستم‌های فایل و ... هر جا که "کل" از "اجزایی" تشکیل شده که خودشان می‌توانند "کل" باشند، کامپوزیت جوابگوی نیاز است.

🎯 **چالش:** داشتن متد مشترک addChild و removeChild به صورت مشترک در Leaf و Composite آن هم به صورتیکه در Leaf هیچ کاری نکند، نقض آشکار Interface Segregate در اصول Solid است!

در پیاده‌سازی کلاسیک الگوی Composite، اگر همه متدهای Component مثل addChild و removeChild را در یک اینترفیس واحد قرار دهیم، آنگاه کلاس‌های Leaf مجبور می‌شوند این متدها را پیاده‌سازی کنند حتی اگر فقط با یک بدنه خالی {} باشد.

این دقیقاً نقض ISP است، چون:

Leaf نیازی به addChild ندارد، اما مجبور است آن را پیاده‌سازی کند، و این منجر به اینترفیس‌های چاق و غیرضروری می‌شود!

💡 راه حل استاندارد این چالش: در منابع معتبر (مثل کتاب GOF و مقالات مدرن طراحی نرم‌افزار)، دو راه‌حل اصلی پیشنهاد شده است:

💡 راه‌حل 1: دو اینترفیس جداگانه (ISP-compliant)

- یک اینترفیس پایه برای همه اجزا (مثل `render`, `getPrice`) داشته باشیم.
- و یک اینترفیس اختیاری برای اجزای مرکب (مثل `IContainer` یا `IComposite`) که فقط `Composite` آن را پیاده‌سازی کند.

```
// اینترفیس پایه - برای همه
class Component {
public:
    virtual ~Component() = default;
    virtual void render() const = 0;
    virtual double getPrice() const = 0;
};

// اینترفیس اختصاصی برای کانتینرها
class IContainer {
public:
    virtual ~IContainer() = default;
    virtual void addChild(std::unique_ptr<Component>) = 0;
    virtual void removeChild(size_t index) = 0;
};

// فقط Leaf Component را پیاده‌سازی می‌کند
class Button : public Component {
public:
    void render() const override { /* ... */ }
    double getPrice() const override { return 0.0; }
};

// Composite هر دو را پیاده‌سازی می‌کند
class Panel : public Component, public IContainer {
private:
    std::vector<std::unique_ptr<Component>> children;
public:
    void render() const override {
        for (auto& child : children)
            child->render();
    }
    double getPrice() const override {
        return std::accumulate(children.begin(), children.end(), 0.0,
            [](double sum, auto& c) { return sum + c->getPrice(); });
    }
    void addChild(std::unique_ptr<Component> c) override {
        children.push_back(std::move(c));
    }
    void removeChild(size_t i) override {
        children.erase(children.begin() + i);
    }
};
```

✅ حالا `Button` هیچ ارتباطی با `addChild` ندارد و `ISP` رعایت شده است.

🎯 اما چالش جدید: کلاینت نمی‌تواند همیشه از `Component` برای اضافه کردن فرزند استفاده کند. پس باید قبل از فراخوانی `addChild`.

بررسی کند که شیء از نوع `IContainer` است!

```
void addElement(Component* parent, std::unique_ptr<Component> child) {
    if (auto container = dynamic_cast<IContainer*>(parent)) {
        container->addChild(std::move(child));
    }
    else {
        std::cerr << "Error: parent is not a container!\n";
    }
}
```

🔦 راه حل 2: پذیرش محدودیت کلاسیک Composite (GoF Style)

در نسخه اصلی GoF، این مشکل عمداً نادیده گرفته می‌شود، چون:

- هدف اصلی یکنواختی رفتار کلاینت است.
- فرض می‌شود کلاینت‌ها به‌درستی طراحی شده‌اند و `addChild` را فقط روی `Composite` فراخوانی می‌کنند.
- در بسیاری از موارد مثل UI ساختار از پیش مشخص است و خطر فراخوانی اشتباه کم است.

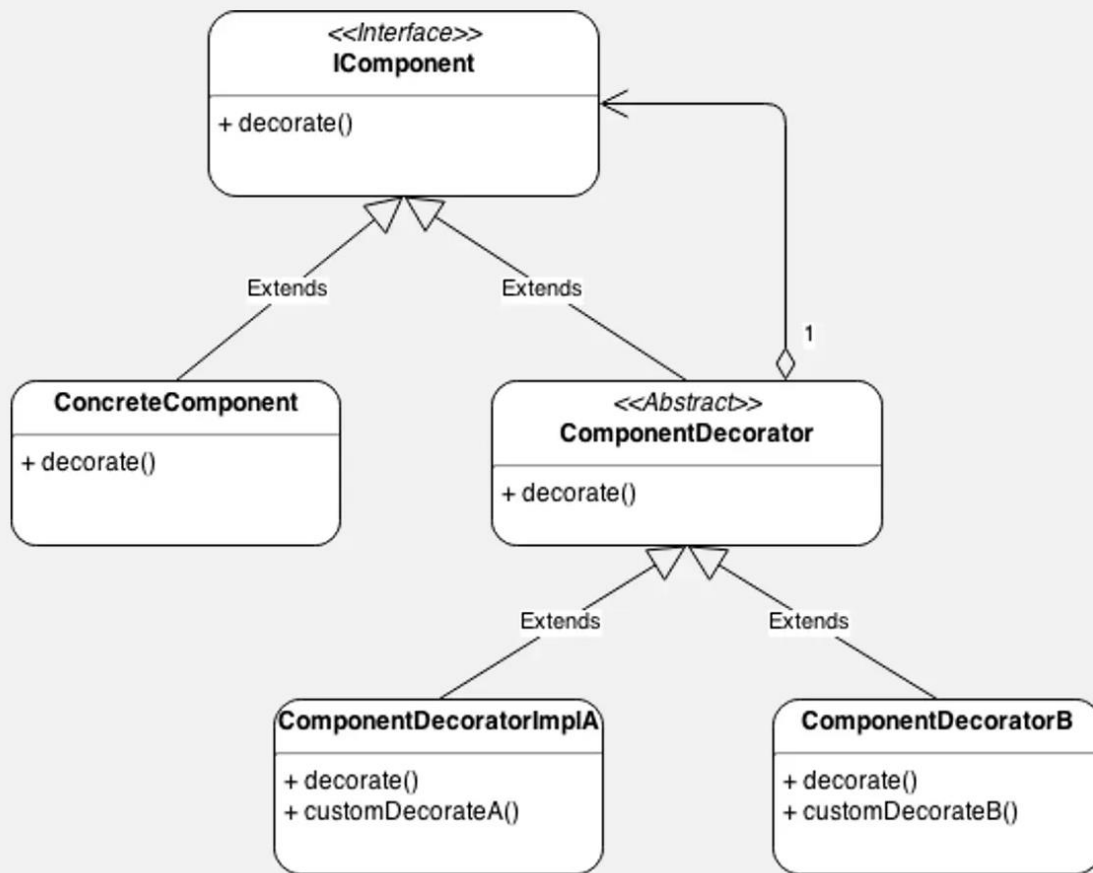
اما اگر در سیستم شما این خطر وجود دارد، راه حل اول (دو اینترفیس) بهتر است!

البته در برخی ساختارها مانند UI که می‌توان یک `Button` را داخل یک `Container` اضافه کرد و همچنین می‌توان درون یک `Button` یک `Label` یا هر چیز دیگری اضافه کرد و در جا به جایی هر کدام از این اقلام، فرزندان‌شان هم همراه با آنها جا به جا می‌شوند، می‌توان گفت ما مفهومی به نام برگ یا `Leaf` نداریم و تمام اشیاء از نوع `Composite` یا `Container` هستند! در نهایت می‌توان گفت بسته به طراحی و نیازمندان، می‌توانید روش مناسبی را انتخاب و بر اساس آن عمل کنید!

Decorator: پترنی از نوع Structural است!



Decorator pattern – Class diagram



این الگوی طراحی تلاش دارد بدون تغییر در اشیای ساخته شده دیگر از یک کلاس، در یک یا چند شی هدف خاص، به صورت استاتیک یا دینامیک یک سری متد و رفتار اضافه کند!

فرض کنید شما کلاس موجودات زنده را دارید و در این میان، کلاس انسان را ساخته اید و از روی این کلاس، چند میلیون شی انسان آفریده اید و فقط و فقط یک انسان خاص به نام سوپرمن وجود دارد که بال دارد و پرواز می کند و انسان دیگری وجود دارد به نام فرانکشتاین که صدایش کاملاً خفه است. پس منطقی نیست رفتار پرواز یا صدای خفه را به تمام انسانها یا موجودات زنده تعمیم دهید. بلکه یک انسان به صورت استثنایی این مشخصه را دارد و بس! حال اهمیت این موضوع در جاییست که قرار است که تعداد این رفتارهای خاص و استثنایی روی انواع اشیا بسیار زیاد باشد و شما بخواهید بین اضافه کردن تمام این رفتارهای استثنایی روی اصل کلاس و اضافه کردن آنها به صورت اختصاصی روی اشیا، یکی را انتخاب کنید و به این نتیجه برسید که دومین انتخاب صحیح است!

```
class IAliveCreature {
public:
    virtual bool Talk(double volume) =0;
    virtual string GetName()=0;
    virtual bool SetName(string name)=0;
};

class Human: public IAliveCreature {
public:
    void Human(string name) {
        SetName(name);
    }
    virtual bool Talk(double volume) {
        if (volume >= 0.0 && volume <= 100.00) {
            std::cout << "I'm "<< GetName() <<" and I'm talking with volume: " << volume <<endl;
            return true;
        }
        return false;
    }
    virtual string GetName() {
        return _name;
    }
    virtual bool SetName(string name){
        if (name != "") {
            _name = name;
            return true;
        }
        return false;
    }
private:
    string _name;
};

class SilentHumanDecorator: public IAliveCreature {
public:
    void SilentHumanDecorator (IAliveCreature* aliveCreaure) {
        _alive_creature = aliveCreaure;
    }
    // تغییر رفتار در صحبت کردن، با هر میزان صدایی، با صدای خفه صحبت می کند
    virtual bool Talk(double volume){
        return _alive_creature->Talk(0);
    }
    virtual string GetName(){
        return _alive_creature->GetName();
    }
};
```

```

    virtual bool SetName(string name){
        return _alive_creature->SetName(name);
    }
private:
    IAliveCreature* _alive_creature;
};

class FlyerHumanDecorator: public IAliveCreature {
public:
    void FlyerHumanDecorator(IAliveCreature* aliveCreaure){
        _alive_creature = aliveCreaure;
    }
    virtual bool Talk(double volume){
        _alive_creature->Talk(volume);
    }
    virtual string GetName(){
        return _alive_creature->GetName();
    }
    virtual bool SetName(string name){
        return _alive_creature->SetName(name);
    }
    //اضافه کردن رفتار پرواز برای سوپر هیومن
    virtual void Fly(){
        std::cout << "Fly With Fluttering My Wings!"<< std::endl;
    }
private:
    IAliveCreature* _alive_creature;
};

int main(){
    Human* normal_human01 = new Human("Hadi");
    Human* normal_human02 = new Human("Hamid");
    Human* normal_human03 = new Human("Mohammad");

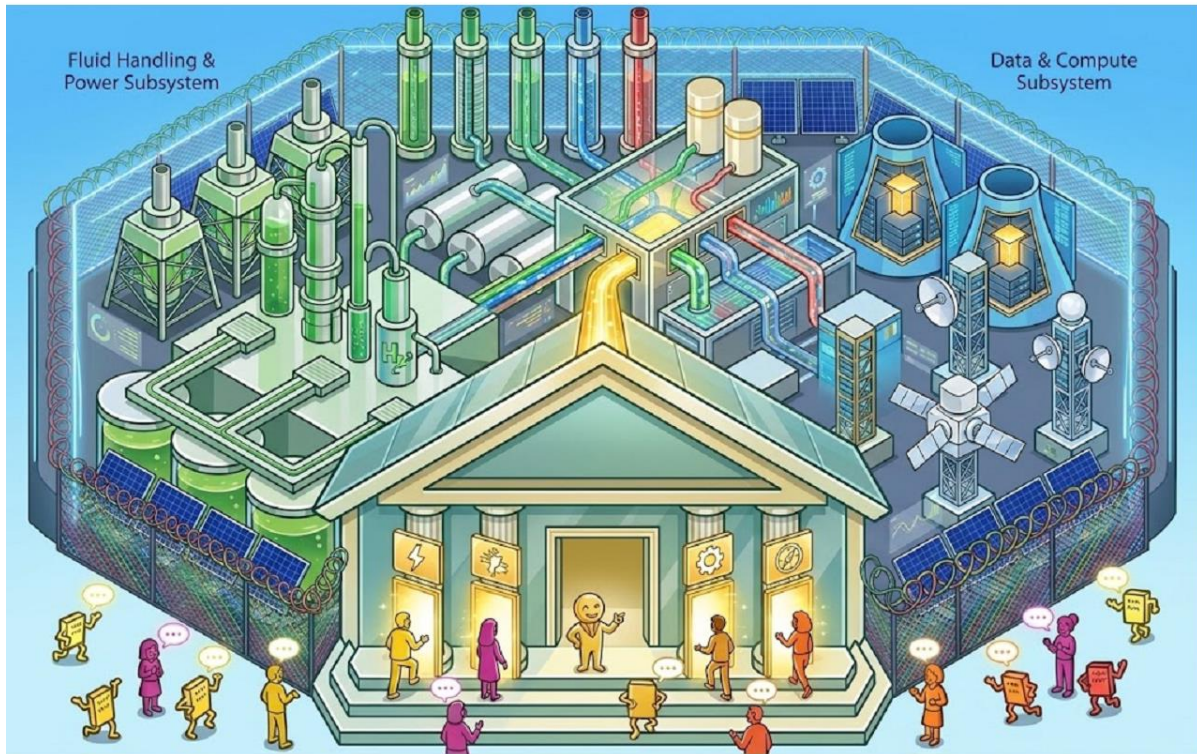
    Human* silent_human = new Human("Frankenstein");
    Human* flyer_human = new Human("super-man");

    SilentHumanDecorator* silent_human_decorator = new
        SilentHumanDecorator(silent_human);
    silent_human_decorator->Talk(20.0); //"I'm silent-man and I'm talking with volume: 0.0
    //silent_human_decorator->Fly(); //Error! there is no any fly method in SilentHumanDecorator
    FlyerHumanDecorator* flyer_human_decorator = new FlyerHumanDecorator(flyer_human);
    flier_human_decorator->Talk(20.0); //"I'm super-man and I'm talking with volume: 20.0
    flier_human_decorator->Fly(); //Fly With Fluttering My Wings!
}

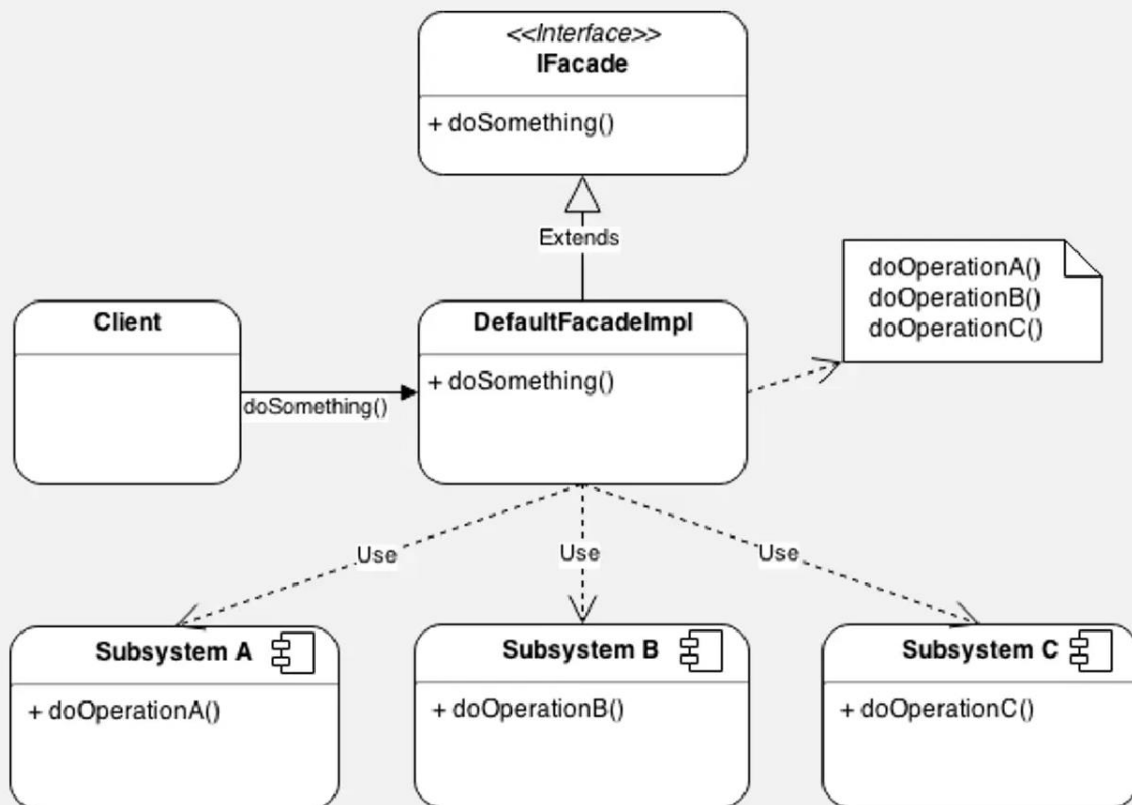
```

پس در اینجا دو کلاس دکوریت یکی برای انسان با صدای خفه و دیگری برای انسان پرنده ساختیم و فقط دو شی را عوض کردیم و به یکی قابلیت جدید و استثنایی پرواز دادیم و رفتار دیگری را تغییر دادیم تا با صدای خفه صحبت کند.

لذا مشاهده کردید که کلاس دکوریتور از اینترفیس کلاس هدفش ارث بری دارد (پیاده سازی می شود) و رفرنسی از آن کلاس را در تابع سازنده دریافت کرده و آنرا در اشاره گر خصوصی از نوع کلاس هدف تنظیم می کند و متدهای اینترفیسی را Override می کند و این Override با صدا زدن متد مد نظر در شی ورودی و سپس تغییر در رفتارهای آن انجام می شود! پس با این روش توانستیم به صورت ران تایم، یا رفتاری جدیدی را بدون تغییر در کلاس اصلی اضافه کنیم و یا بدون تغییر در کلاس اصلی، رفتاری را عوض کردیم!



Facade pattern – Class diagram



فرض کنید قرار است کلاسی را دست کاربر برنامه ساز دهید که یک یا چند **operation** را با استفاده از متدهایی مثل **Operation01** و **Operation02** و **Operation03** انجام دهد. منتها هر کدام از این توابع ترکیبی هستند از عملیات یک، دو یا هزاران تابع یا کلاس جدا از هم که هر کدام وظیفه خاصی را انجام می دهند! به بیان بهتر شما قرار است با استفاده از یک کلاس سیستمی، از چندین زیرسیستم (**Subsystem**) سرویس بگیریم و با پوشاندن عملکرد آنها، به صورت یکپارچه و با ثبات، سرویس نهایی را به کاربر تقدیم نماییم! هدف این عملیات این است که به جای درگیر شدن کاربر با توابع و کلاسهای گوناگون، با استفاده از یک متد و ورود اطلاعات مورد نیاز آن عملیات، آنرا به سادگی برای وی انجام دهیم. به نوعی این دیزاین پترن شبیه به **wrapper** و به نوعی شبیه به **Adapter** است. این کلاس یک سیستم است که یک یا چندین زیر سیستم دارد و شما با استفاده از متدهای **Operation** می خواهید از زیرسیستم های این سیستم، نتیجه صحیح را به شکل یک سیستم جامع و پیوسته بگیرید!

فرض کنید ما یک سیستم خانه هوشمند داریم و می خواهیم با زیرسیستم های داخل آن، به کاربر، خدمات مورد نیازش را ارائه دهیم!

```
#include <string>
#include <iostream>
using namespace std;
// چراغ
class Light {
public:
    void lightOn(){
        cout << "Light is on"<<endl;
    }
    void lightOff(){
        cout << "Alarm is off"<<endl;
    }
};
// دزدگیر
class Alarm {
public:
    void alarmOn(){
        cout << "Alarm is on and house is secured"<<endl;
    }
    void alarmOff(){
        cout << "Alarm is off and you can go into the house"<<endl;
    }
};
// کولر
class AirCondition {
public:
    void acOn(){
        cout << "Ac is on"<<endl;
    }
    void acOff(){
        cout << "AC is off"<<endl;
    }
};
// تلویزیون
class Tv {
public:
    void tvOn(){
        cout << "Tv is on"<<endl;
    }
};
```



```

void tvOff(){
    cout << "TV is off"<<endl;
}

};
// منزل ما که با زیرسیستم های تعریف شده بالا، سرویس نهایی را به کاربر ارائه می دهد
class HouseFacade{
    Alarm* _alarm;
    AirCondition* _ac;
    Tv* _tv;
    Light* _light;
public:
    HouseFacade(Alarm* alarm,
                AirCondition* ac,
                TV* tv,
                Light* light):
        _alarm(alarm),
        _ac(ac),
        _tv(tv),
        _light(light){}

    // Facade
    void goToWork() { // دکمه رفتن سر کار
        _ac->acOff();
        _tv->tvOff();
        _light->lightOff();
        _alarm->alarmOn();
    }

    void comeHome() { // دکمه رفتن به خانه
        _alarm->alarmOff();
        _ac->acOn();
        _light->lightOn();
        _tv->tvOn();
    }

};

int main() {
    HouseFacade hf(new Alarm(),
                   new AirCondition(),
                   new TV(),
                   new Light()
                   );

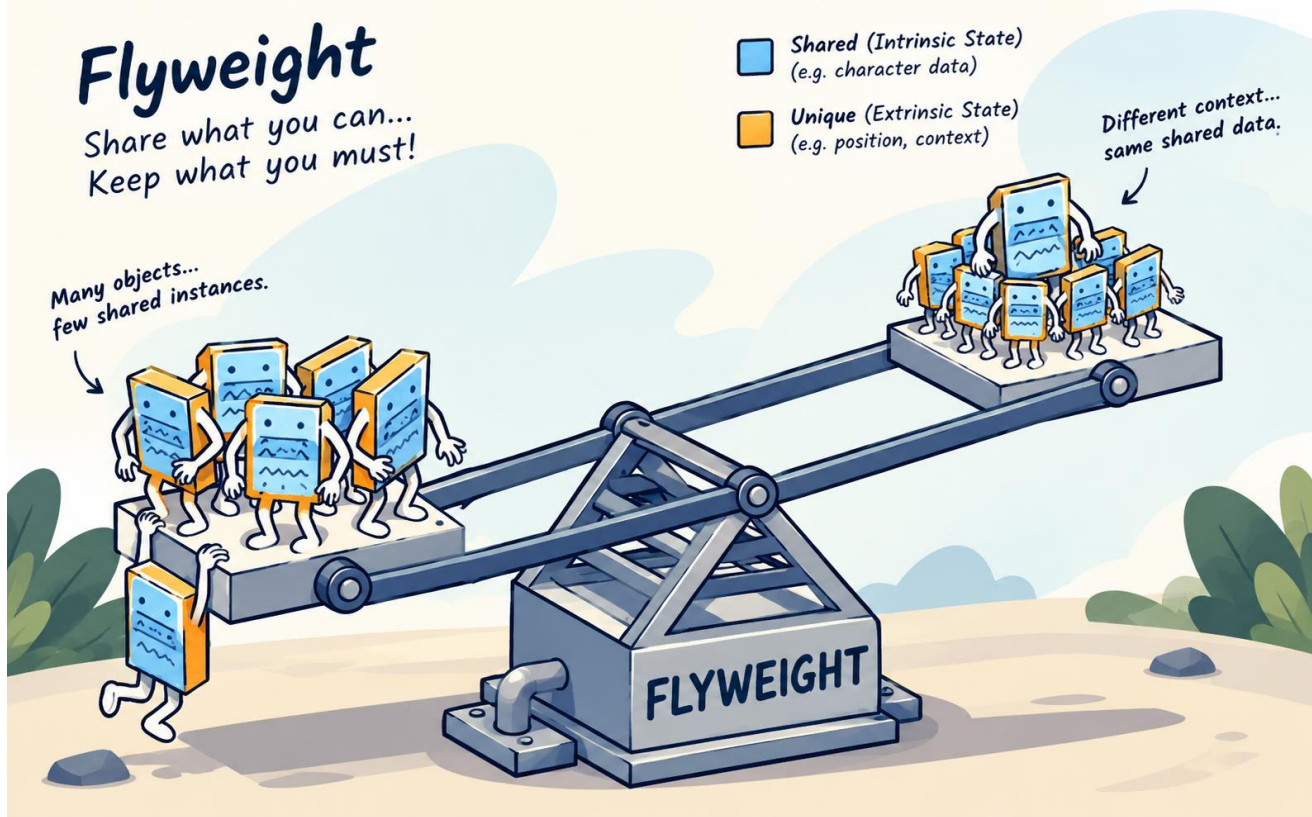
    // به جای اینکه ده ها دکمه روشن/خاموش را بزنیم، با فشار دو دکمه می توانیم به خواست خود برسیم
    hf.goToWork();
    hf.comeHome();
}

```

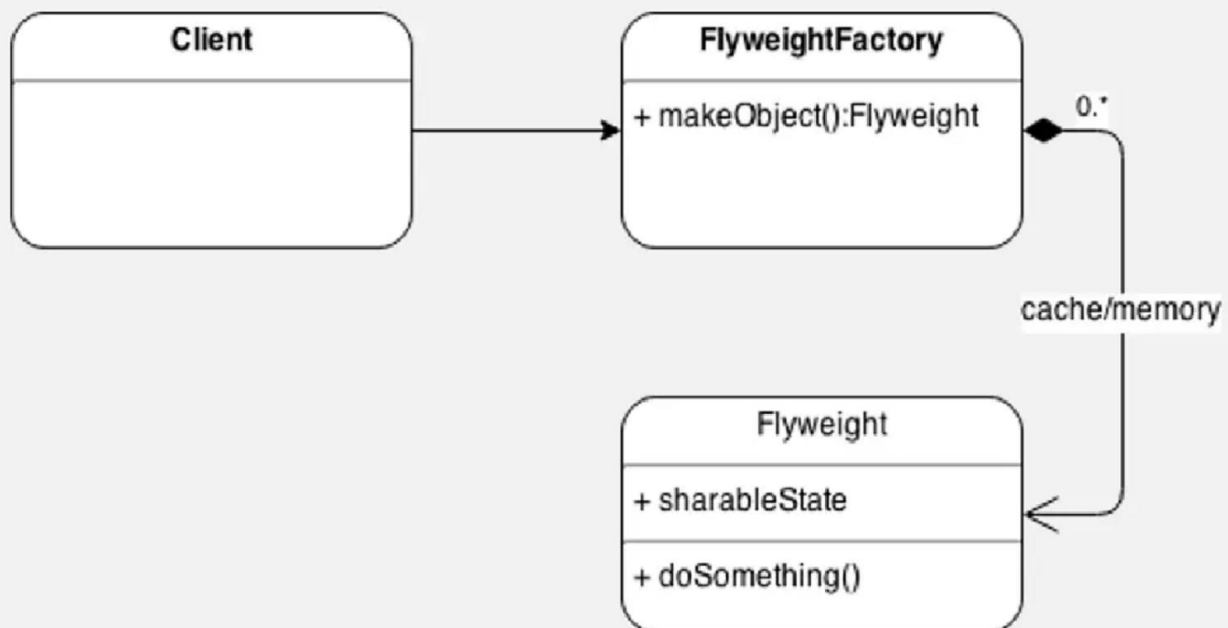
 **نکته در مورد الگوی Facade:** لزوماً توابع Facade با انجام عملیات به صورت Void به ما سرویس ارائه نمی دهند! گاهی ممکن است نتیجه محاسبه ای که از دل چندین Sub System خارج می شود، توسط تابع Facase برای ما return شود! برای مثال فرض کنید شی زیرسیستم احراز هویت بخواهد صحت هویت کاربر را چک کند و اگر true بود، زیرسیستم واکنشی اطلاعات کاربر، اطلاعات وی را واکنشی کند و سرویس نمایش دهنده آنرا نمایش دهد! اگر هم false بود، زیر سیستم تولید خطا، متن یا کد خطایی را تولید کند و نمایش دهنده، اطلاعات خطا را نمایش دهد!

تفاوت Adapter با Decorator و Facade :

الگو	هدف
Adapter	سازگار کردن دو API ناسازگار بدون تغییر کد اصلی
Decorator	اضافه کردن قابلیت بدون تغییر در کلاس اصلی
Facade	ترکیب و ساده سازی چند سیستم پیچیده در قالب یک سیستم یکپارچه



Flyweight pattern – Class diagram



الگوی Flyweight (پترن فلابویت) یک الگوی ساختاری (Structural) است که برای کاهش مصرف حافظه با اشتراک گذاری بخش‌های مشترک بین اشیاء مختلف استفاده می‌شود. این الگو زمانی مفید است که نیاز به ایجاد تعداد زیادی شیء داریم که بسیاری از ویژگی‌هایشان مشترک است. و اما مشکلی که Flyweight حل می‌کند ...!

فرض کنید می‌خواهید ۱۰۰۰۰۰ نقطه روی نقشه جغرافیایی نمایش دهید!

- هر نقطه دارای مختصات X, Y, نوع (کافه، ایستگاه، ...)، تصویر، رنگ، اندازه و... است!

- بدون Flyweight ۱۰۰۰۰۰ شیء × همراه با همه ویژگی‌ها را دارید که مصرف حافظه را بسیار بالا می‌برد!

- با Flyweight ویژگی‌های ذاتی (Intrinsic) را به اشتراک می‌گذاریم که مصرف حافظه را پایین می‌آوریم!

✖ 1- حالت ذاتی (Intrinsic State)

- ویژگی‌هایی که بین اشیاء مشترک هستند!

- قابل اشتراک گذاری بین چندین شیء

🎨 مثال: تصویر، رنگ، اندازه (برای یک نوع خاص مثلاً آیکن کافه‌های روی یک نقشه)

✖ 2- حالت بیرونی (Extrinsic State)

- ویژگی‌های منحصر به فرد هر شیء

- غیرقابل اشتراک گذاری

🎨 مثال: مختصات X و Y

✖ پیاده‌سازی مثال نقاط نقشه

```
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>
#include <memory>
// مسیر یا داده‌های تصویر (فقط برای نمایش)
struct Image {
    std::string data;
};
// Enum برای انواع نقاط
enum PointType {
    CAFE,
    STATION,
    RESTAURANT,
    HOSPITAL,
    SCHOOL
};
```

```

// تبدیل پوینت تایپ ها به اسامی معادل
std::string pointTypeToString(PointType type) {
    switch (type) {
        case CAFE: return "CAFE";
        case STATION: return "STATION";
        case RESTAURANT: return "RESTAURANT";
        case HOSPITAL: return "HOSPITAL";
        case SCHOOL: return "SCHOOL";
        default: return "UNKNOWN";
    }
}

// Flyweight Interface
class PointIcon {
public:
    virtual ~PointIcon() = default;
    virtual PointType getType() const = 0;
    virtual const Image& getIcon() const = 0;
};

class PointIconImpl : public PointIcon { // Concrete Flyweight
private:
    PointType type;
    std::shared_ptr<Image> icon; // استفاده از اشاره گر اسمارت برای مدیریت اتوماتیک
    std::string color; // رنگ مشترک (Intrinsic)
    int size; // سایز مشترک (Intrinsic)
public:
    PointIconImpl(PointType type,
                  std::shared_ptr<Image> icon,
                  std::string color,
                  int size)
        : type(type),
          icon(std::move(icon)),
          color(std::move(color)),
          size(size) {}
    PointType getType() const override {
        return type;
    }
    const Image& getIcon() const override {
        return *icon;
    }
};

// Flyweight Factory
class PointIconFactory {
private:
    // استفاده از shared_ptr برای اینکه اشیاء بین Point ها به اشتراک گذاشته شوند
    std::unordered_map<PointType, std::shared_ptr<PointIcon>> icons;
    std::shared_ptr<Image> loadIconForType(PointType type) {
        std::cout << "Loading icon for " << pointTypeToString(type) << "\n";
        auto img = std::make_shared<Image>();
        //img->data = loaded_data from type; // لود فایل تصویر از روی نوع آیکن
        return img; // در واقعیت تصویر واقعی لود می شود
    }
    std::string getColorForType(PointType type) {
        switch (type) {
            case CAFE: return "Brown";
            case STATION: return "Blue";
            case RESTAURANT: return "Red";
            case HOSPITAL: return "Green";
            case SCHOOL: return "Yellow";
            default: return "Black";
        }
    }
}

```

```

int getSizeForType(PointType type) {
    return 32; // سائز یکسان برای همه
}
public:
    std::shared_ptr<PointIcon> getPointIcon(PointType type) {
        // اگر از قبل وجود نداشته، آن را بساز
        if (icons.find(type) == icons.end()) {
            auto icon = loadIconForType(type);
            std::string color = getColorForType(type);
            int size = getSizeForType(type);
            auto pointIcon = std::make_shared<PointIconImpl>(type,
                                                            icon,
                                                            color,
                                                            size);

            icons[type] = pointIcon;
        }
        // در این مرحله می‌دانیم که اگر نبوده ساخته‌ایم و اگر بوده که هیچ
        return icons[type];
    }
};
// Context کلاسی که State بیرونی یا Extrinsic را نگه می‌دارد
class Point {
private:
    int x; // Extrinsic state (منحصر به فرد)
    int y; // Extrinsic state (منحصر به فرد)
    std::shared_ptr<PointIcon> icon; // Intrinsic state (مشترک)

public:
    Point(int x, int y, std::shared_ptr<PointIcon> icon)
        : x(x), y(y), icon(std::move(icon)) {}

    void draw() const {
        std::cout << "Drawing " << pointTypeToString(icon->getType())
                  << " at (" << x << ", " << y << ") with shared icon\n";
    }
    int getX() const { return x; }
    int getY() const { return y; }
    PointType getType() const { return icon->getType(); }
};
// Client / Container
class Map {
private:
    std::vector<Point> points;
    PointIconFactory iconFactory;

public:
    void addPoint(int x, int y, PointType type) {
        // دریافت Flyweight از Factory
        auto icon = iconFactory.getPointIcon(type);
        // ایجاد نقطه با مختصات منحصر به فرد
        points.emplace_back(x, y, icon);
    }
    void drawAll() const {
        for (const auto& point : points) {
            point.draw();
        }
    }
    int getPointCount() const {
        return points.size();
    }
};

```

```

int main() {
    Map map;
    // اضافه کردن ۵ نقطه کافه (اشتراک گذاری یک Flyweight)
    map.addPoint(10, 20, CAFE);
    map.addPoint(15, 25, CAFE);
    map.addPoint(20, 30, CAFE);
    map.addPoint(25, 35, CAFE);
    map.addPoint(30, 40, CAFE);
    // اضافه کردن ۳ نقطه ایستگاه
    map.addPoint(100, 200, STATION);
    map.addPoint(150, 250, STATION);
    map.addPoint(200, 300, STATION);
    // اضافه کردن نقاط مختلف دیگر
    map.addPoint(50, 60, RESTAURANT);
    map.addPoint(70, 80, HOSPITAL);

    std::cout << "Total points on map: " << map.getPointCount() << "\n";
    std::cout << "\nDrawing all points:\n";
    map.drawAll();

    return 0;
}

```

مزایای الگوی Flyweight

- صرفه جویی در حافظه: کاهش مصرف تا ۹۰٪ در سناریوهای مناسب
- بهبود عملکرد: کاهش زمان لود تصاویر و منابع
- مدیریت متمرکز: مدیریت اشیاء مشترک در یک مکان
- مقیاس پذیری: توانایی ایجاد میلیون ها شیء با منابع محدود

معایب و محدودیت ها:

- پیچیدگی کد: افزایش پیچیدگی طراحی
- سر بار محاسباتی: هزینه جستجو در Factory
- مناسب نبودن برای همه موارد: فقط وقتی مفید است که:

- تعداد زیادی شیء ایجاد می کنید!

- اشیاء حالت ذاتی مشترک دارند!

- حالت بیرونی را می توان جدا کرد!

سناریوهای عملی استفاده:

1. ویرایشگرهای متن - کاراکترها (فونت، سایز مشترک)
2. بازی های کامپیوتری - درختان، ساختمان ها، واحدهای نظامی
3. نقشه های تعاملی - نقاط، آیکون ها

4. سیستم‌های گرافیکی - اشکال هندسی تکراری

5. برنامه‌های طراحی - سمبل‌ها و ابزارها

👉 مقایسه با و بدون Flyweight:

✗ بدون Flyweight (خام)

```
// هر Point همه چیز را خودش ذخیره می‌کند
public class NaivePoint {
    private int x, y;
    private PointType type;
    private Image icon;
    private String color;
    private int size;

    // ۱۰۰۰۰۰ شیء = ۱۰۰۰۰۰ تصویر در حافظه!
}
```

✓ با Flyweight (بهینه)

100000 نقطه شامل مختصات X, Y و همچنین اشاره گر نقاط فقط به 5 عدد Flyweight!

📊 محاسبه صرفه‌جویی حافظه: فرض کنید هر شیء سایزهای زیر را دارد:

تصویر 10KB

رنگ: 20B

سایز: 4B

مختصات: 8B

بدون Flyweight (100,000 نقطه):

$$(10KB + 20B + 4B + 8B) \times 100,000 \approx 1GB$$

با Flyweight (5 نوع مختلف):

فقط 5 نوع فلاینیت ها Flyweights: $(10KB + 20B + 4B) \times 5 \approx 50KB$

نقاط:

$$8B \times 100,000 \approx 0.8MB \quad \text{فقط نقاط همراه با مختصات}$$

کل:

کل تقریباً 0.85MB

به بیان بهتر یعنی با FlyWeight ما حدود 99 درصد صرفه جویی در حافظه داشتیم! البته باید در نظر گرفت هر نقطه غیر از مختصات، یک شی از نوع Type هم دارد که باز هم با افزودن Type فرق آنچنان خاصی در مصرف حافظه نخواهد داشت!

🚀 نکات پیشرفته:

1- Thread Safety

```
#include <unordered_map>
#include <memory>
#include <shared_mutex>
#include <string>
// پیش‌تعریف کلاس‌ها و تایپ‌های مورد نیاز (بسته به پیاده‌سازی واقعی شما)
// enum class PointType;
// class PointIcon;
// class PointIconImpl;
// class Image;
class ThreadSafePointIconFactory {
private:
    std::unordered_map<PointType, std::shared_ptr<PointIcon>> icons;

    mutable std::shared_mutex mtx; // برای خواندن همزمان چند رشته
    // نیازمند (C++17)
    std::shared_ptr<PointIcon> createIcon(PointType type) {
        // ایجاد flyweight جدید
        return std::make_shared<PointIconImpl>(type,
                                                loadIcon(type),
                                                getColor(type),
                                                getSize(type));
    }
    // این متدها باید در کلاس یا بیرون از آن تعریف شده باشند
    Image loadIcon(PointType type);
    std::string getColor(PointType type);
    int getSize(PointType type);
public:
    std::shared_ptr<PointIcon> getPointIcon(PointType type) {
        // ۱. ابتدا با قفل اشتراکی (Shared Lock) جستجو می‌کنیم (سرعت بالا)
        {
            std::shared_lock<std::shared_mutex> readLock(mtx);
            auto it = icons.find(type);
            if (it != icons.end()) {
                return it->second;
            }
        }
        // ۲. اگر وجود نداشت، قفل انحصاری (Unique Lock) می‌گیریم
        std::unique_lock<std::shared_mutex> writeLock(mtx);
        // بررسی مجدد (Double-Checked Locking) برای اطمینان از اینکه رشته دیگری آن/
        // را ایجاد نکرده باشد
        auto it = icons.find(type);
        if (it != icons.end()) {
            return it->second;
        }
        // ایجاد و ذخیره معادل computeIfAbsent در جاوا
        auto icon = createIcon(type);
        icons.emplace(type, icon);
        return icon;
    }
};
```

چرا در `getPointIcon` از دو قفل تردی استفاده کردیم؟ قفل دوم فقط در شرایطی اتفاق می افتد که کلید خواسته شده ساخته نشده باشد، ولی به محض اینکه آبجکت ساخته شد، تمام درخواست‌های بعدی (که ۹۹٪ مواقع همین‌طور است) در همان لاک اشتراکی اول (که بسیار سریع است و اجازه خواندن هم‌زمان به همه می‌دهد) جواب می‌گیرند و اصلاً به لاک دوم نمی‌رسند. بنابراین هزینه لاک انحصاری فقط "یک بار" برای هر `PointType` پرداخت می‌شود. قفل دوم به چه دردی می‌خورد؟ اگر دو ترد موازی وارد تابع `getPointIcon` شوند، تردی که زودتر وارد آکولاد حوزه قفل اول می‌شود، زودتر آن قفل را می‌بندد تا دومی وارد آن آکولاد نشود! حالا فرض کنید آیکونی را سرچ می‌کند و در مپ مدنظر آنرا نمی‌بیند و از بلاک قفل اول بیرون می‌آید و قفل اول را آزاد می‌کند و در کسری از ثانیه ترد دوم هم با بررسی همان آیکون، وقتی آن آیکون را ندید، با خروج از بلاک قفل اول، قفل اول را برای درخواست‌های بعدی باز می‌کند! فرض کنید قفل دوم نباشد، هر دو ترد هم در این مرحله، از قفل اول عبور کرده‌اند و در این مرحله اول، ترد اول، آیکن مد نظر خود را که در سرچ محدوده قفل اول آنرا نیافته، آنرا بسازد و در `map` مربوطه قرار دهد! سپس (یا شاید قبل از ترد اول) ترد دوم آن آیکون را دوباره بسازد و در همان `map` و در همان کلیدی که ترد اول آن کلید را دنبال می‌کرد و نهایتاً ساخته، دوباره قرار داده باشد! اینجا قفل دوم به کار می‌آید تا اجازه ندهد آیکنی که هر دو ترد نیاز به آن دارند، 2 بار ساخته شود! به صورت کلی خاصیت قفل اشتراکی آنست که در زمان خواندن، به همه اجازه می‌دهد آنرا بخواند (و از دروازه قفل اشتراکی رد شوند) ولی در هنگام نوشتن، با `unique_lock` کردن، حتی تردهای قبل از قفل اشتراکی هم قفل می‌شوند!

2- Flyweight با حالت تغییرپذیر

```
#include <string>
#include <memory>
#include <mutex>

class MutableFlyweight {
private:
    std::shared_ptr<Image> icon;
    // اگر Image یک کلاس است، از shared_ptr استفاده می‌کنیم
    std::string color;
    mutable std::mutex mtx; // برای امنیت رشته‌ها در صورت تغییر همزمان
public:
    MutableFlyweight(std::shared_ptr<Image> icon,
                    std::string color)
        : icon(std::move(icon)),
          color(std::move(color)) {}

    // اگر نیاز به تغییر داشت:
    void updateColor(const std::string& newColor) {
        std::lock_guard<std::mutex> lock(mtx);
        this->color = newColor;
        // همه نقاطی که از این Flyweight استفاده می‌کنند به‌روز می‌شوند
    }

    std::string getColor() const {
        std::lock_guard<std::mutex> lock(mtx);
        return color;
    }
};
```

۳. ترکیب با الگوهای دیگر

– Factory Method برای ایجاد Flyweight ها

– Singleton برای Factory

– Composite برای ساختارهای سلسله‌مراتبی

 جمع بندی: الگوی Flyweight یک ابزار قدرتمند بهینه‌سازی حافظه است که:

1. حالت ذاتی (مشترک) و حالت بیرونی (منحصر به فرد) را جدا می‌کند!

2. از یک Factory برای مدیریت اشتراک‌گذاری استفاده می‌کند!

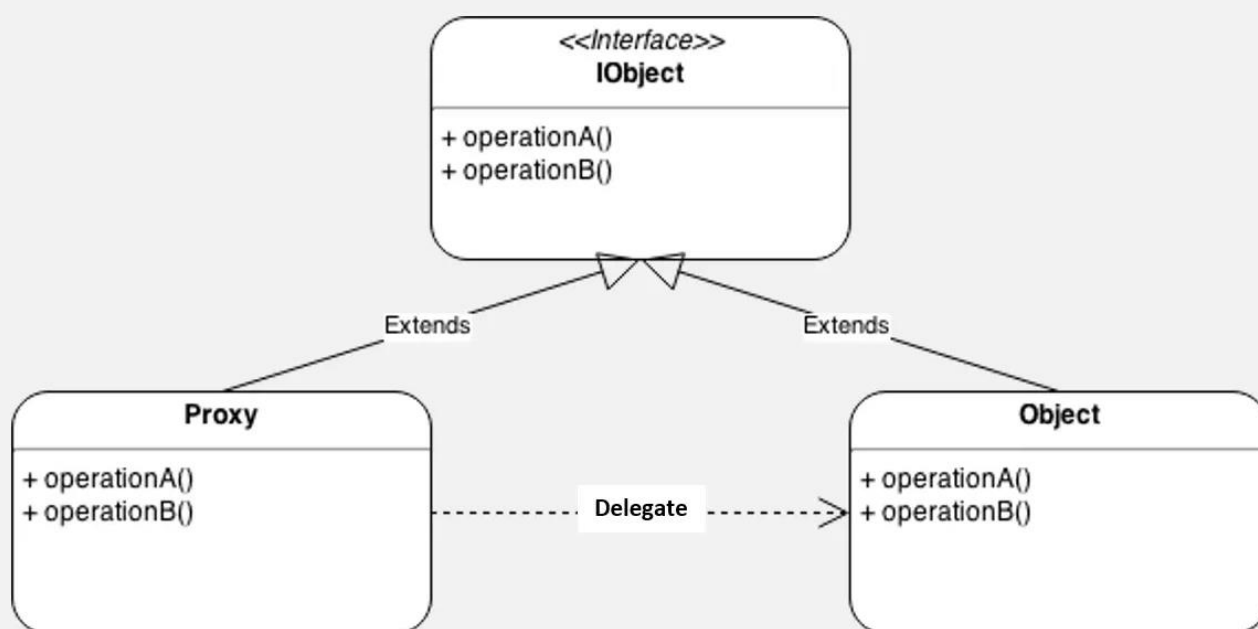
3. برای اشیاء با تعداد زیاد و ویژگی‌های مشترک ایده‌آل است!

4. در سیستم‌های گرافیکی و بازی‌ها کاربرد فراوان دارد!

این الگو در مثال نقاط نقشه به ما اجازه داد میلیون‌ها نقطه را با حداقل حافظه مدیریت کنیم، در حالی که هر نقطه مختصات منحصر به فرد خود را دارد اما خصوصیات ظاهری مشترک را به اشتراک می‌گذارد!



Proxy pattern – Class diagram



پترن Proxy یک الگوی طراحی ساختاری است که به شما امکان می‌دهد یک جایگزین یا نماینده برای شیء دیگری ایجاد کنید. این الگو به شما امکان کنترل دسترسی به شیء اصلی را می‌دهد.

 **کل ایده:** هم کلاس اصلی و هم کلاس نماینده از یک Interface خاص پیاده سازی می‌شوند منتها کلاس پروکسی یا نماینده، یک شیء از نوع کلاس اصلی در دل خود دارد و با کال کردن متدهای هم نام در سطح پراکسی، متدهای هم نام در شیء اصلی با ملاحظات صد زده می‌شود! پس به جای اینکه مستقیم با آبجکت اصلی کار کنیم، از Proxy استفاده کنیم!

انواع Proxy:

1. **Virtual Proxy:** برای lazy initialization

2. **Protection Proxy:** برای کنترل دسترسی

3. **Remote Proxy:** برای اشیاء در مکان‌های دور

4. **Smart Proxy:** برای عملیات اضافه مانند reference counting

ساختار پایه Proxy

```
#include <iostream>
#include <string>
#include <memory>
#include <vector>
#include <unordered_map>
#include <thread>
#include <chrono>
#include <mutex>
// مشترک Interface
class EBook {
public:
    virtual ~EBook() = default;
    virtual void display() const = 0;
    virtual std::string getTitle() const = 0;
    virtual std::string getAuthor() const = 0;
    virtual int getPageCount() const = 0;
    virtual void loadContent() = 0;
    virtual bool isLoading() const = 0;
};
```

 **مثال ۱:** Virtual Proxy برای Lazy Loading کتاب‌های الکترونیکی

```
// کتاب واقعی - Real Subject
class RealEBook : public EBook {
private:
    std::string title;
    std::string author;
    int pageCount;
    std::string content;
    bool loaded;
```

```

// شبیه‌سازی بارگذاری سنگین
void simulateHeavyLoading() {
    std::cout << "📖 در حال بارگذاری کتاب << title << "...\n";
    std::this_thread::sleep_for(std::chrono::seconds(2));

    // تولید محتوای نمونه
    content = "است.\n" + title + " اثر " + author + " این محتوای کامل کتاب";
    content += "صفحه دارد " + std::to_string(pageCount) + " این کتاب";
    content += "محتوا از پایگاه داده یا فایل بارگذاری شده است.\n";

    loaded = true;
    std::cout << "✅ کتاب " << title << " با موفقیت بارگذاری شد.\n";
}

public:
    RealEBook(const std::string& title, const std::string& author, int pages)
        : title(title), author(author), pageCount(pages), loaded(false) {
        std::cout << "📖 برای RealEBook شیء " << title << " ایجاد شد.\n";
    }

    void display() const override {
        if (!loaded) {
            std::cout << "⚠️ هنوز بارگذاری نشده است " << title << "\n";
            return;
        }

        std::cout << "\n=== " << title << " ===\n";
        std::cout << "نویسنده: " << author << "\n";
        std::cout << "تعداد صفحات: " << pageCount << "\n";
        std::cout << "محتوا:\n" << content << "\n";
    }

    std::string getTitle() const override { return title; }
    std::string getAuthor() const override { return author; }
    int getPageCount() const override { return pageCount; }

    void loadContent() override {
        if (!loaded) {
            simulateHeavyLoading();
        }
    }

    bool isLoading() const override { return loaded; }
};

// Proxy - نماینده کتاب
class EBookProxy : public EBook {
private:
    std::string title;
    std::string author;
    int pageCount;
    mutable std::unique_ptr<RealEBook> realBook; // Lazy initialization
    mutable std::mutex loadMutex;
    // دسترسی به RealEBook با Lazy Loading
    RealEBook* getRealBook() const {
        std::lock_guard<std::mutex> lock(loadMutex);
        if (!realBook) {
            realBook = std::make_unique<RealEBook>(title, author, pageCount);
        }
        return realBook.get();
    }
}

```

```

public:
    EBookProxy(const std::string& title, const std::string& author, int pages)
        : title(title), author(author), pageCount(pages) {
        std::cout << "📖 Proxy برای کتاب '" << title << "' ایجاد شد.\n";
    }
    void display() const override {
        std::cout << "\n[Proxy] درخواست نمایش کتاب '" << title << "'\n";
        if (!realBook || !realBook->isLoaded()) {
            std::cout << "[Proxy] فقط اطلاعات اولیه نشده.\n";
            std::cout << "عنوان: " << title << "\n";
            std::cout << "نویسنده: " << author << "\n";
            std::cout << "تعداد صفحات: " << pageCount << "\n";
            std::cout << "[Proxy] بارگذاری کنید.\n";
        } else {
            realBook->display();
        }
    }
    std::string getTitle() const override { return title; }
    std::string getAuthor() const override { return author; }
    int getPageCount() const override { return pageCount; }

    void loadContent() override {
        std::cout << "[Proxy] بارگذاری کتاب '" << title << "'\n";
        getRealBook()->loadContent();
    }

    bool isLoaded() const override {
        return realBook && realBook->isLoaded();
    }
    void unload() { // متد اضافی برای آزاد کردن حافظه (برای مدیریت بهتر)
        std::lock_guard<std::mutex> lock(loadMutex);
        if (realBook && realBook->isLoaded()) {
            std::cout << "[Proxy] کتاب '" << title << "' از حافظه آزاد شد.\n";
            realBook.reset();
        }
    }
};

class DigitalLibrary { // 4. Library - کلاس استفاده‌کننده
private:
    std::vector<std::shared_ptr<EBook>> ebooks;
    std::unordered_map<std::string, std::shared_ptr<EBook>> ebookCache;

public:
    void addEBook(const std::shared_ptr<EBook>& ebook) {
        ebooks.push_back(ebook);
        ebookCache[ebook->getTitle()] = ebook;
    }

    void browseLibrary() const {
        std::cout << "\n📖 === کتابخانه دیجیتال ===\n";
        std::cout << "تعداد کتابها: " << ebooks.size() << "\n\n";

        for (size_t i = 0; i < ebooks.size(); ++i) {
            std::cout << i + 1 << ". " << ebooks[i]->getTitle()
                << " - " << ebooks[i]->getAuthor()
                << " (" << ebooks[i]->getPageCount() << " صفحه)\n";
        }
    }
}

```

```

void readEBook(const std::string& title) {
    auto it = ebookCache.find(title);
    if (it != ebookCache.end()) {
        std::cout << "\n📖 انتخاب کتاب: " << title << "\n";
        // فقط نمایش اولیه بدون بارگذاری
        it->second->display();
        // درخواست بارگذاری از کاربر
        std::cout << "\n؟ کنید (y/n): ";
        char choice;
        std::cin >> choice;

        if (choice == 'y' || choice == 'Y') {
            it->second->loadContent();
            it->second->display();
        }
    } else {
        std::cout << "❌ کتاب '" << title << "' یافت نشد.\n";
    }
}

void preloadAll() {
    std::cout << "\n📖 در حال بارگذاری همه کتابها\n";
    for (auto& ebook : ebooks) {
        ebook->loadContent();
    }
}

};
// مثال اجرا
int main() {
    std::cout << "=== Virtual Proxy برای Lazy Loading EBooks ===\n\n";

    // ایجاد کتابخانه
    DigitalLibrary library;
    // Proxy اضافه کردن کتابها با استفاده از
    // هنوز ایجاد نشده است RealEBook: توجه
    library.addEBook(std::make_shared<EBookProxy>(
        "طراحی الگوها", "اریک گاما", "395"));

    library.addEBook(std::make_shared<EBookProxy>(
        "کلین تمیز", "رابرت مارتین", "464"));

    library.addEBook(std::make_shared<EBookProxy>(
        "الگوریتمها", "توماس کورمن", "1312"));
    // مرور کتابخانه
    library.browseLibrary();
    // خواندن کتاب اول (بدون بارگذاری)
    std::cout << "\n📖 مشاهده کتاب اول بدون بارگذاری\n";
    library.readEBook("طراحی الگوها");
    // خواندن کتاب دوم (با بارگذاری)
    std::cout << "\n📖 مشاهده کتاب دوم با بارگذاری\n";
    library.readEBook("طراحی الگوریتم");
    // نمایش وضعیت
    std::cout << "\n📊 وضعیت فعلی\n";

    auto proxy1 =
        std::dynamic_pointer_cast<EBookProxy>(library.getEBook("طراحی الگوها"));
}

```



```

if (proxy1) {
    std::cout << "کتاب 'طراحی الگوها' بارگذاری شده؟"
                << (proxy1->isLoading() ? "✅ خیر ❌" : "بله") << "\n";
}
return 0;
}

```

مثال 2: Protection Proxy برای کنترل دسترسی

```

// اینترفیس برای کاربران
class User {
public:
    virtual ~User() = default;
    virtual std::string getRole() const = 0;
    virtual std::string getName() const = 0;
};

// کاربر واقعی
class RealUser : public User {
private:
    std::string name;
    std::string role;
public:
    RealUser(const std::string& name, const std::string& role)
        : name(name), role(role) {}

    std::string getRole() const override { return role; }
    std::string getName() const override { return name; }
};

// EBook برای Protection Proxy
class ProtectedEBook : public EBook {
private:
    std::shared_ptr<RealEBook> realEBook;
    std::shared_ptr<User> user;

    bool hasAccess() const {
        // فقط کاربران Premium یا Admin می‌توانند به کتاب دسترسی داشته باشند
        return user->getRole() == "Premium" || user->getRole() == "Admin";
    }
public:
    ProtectedEBook(const std::shared_ptr<RealEBook>& ebook,
                  const std::shared_ptr<User>& user)
        : realEBook(ebook), user(user) {}

    void display() const override {
        if (hasAccess()) {
            std::cout << "👤 کاربر " << user->getName()
                      << " (" << user->getRole() << ") دسترسی دارد.\n";
            realEBook->display();
        } else {
            std::cout << "🚫 دسترسی کاربر رد شد! " << user->getName()
                      << " (" << user->getRole() << ") اجازه مشاهده این کتاب را"
                      << "\n";
            std::cout << "💡 لطفاً اشتراک پریمیوم خریداری نمایید!\n";
        }
    }

    std::string getTitle() const override { return realEBook->getTitle(); }
    std::string getAuthor() const override { return realEBook->getAuthor(); }
    int getPageCount() const override { return realEBook->getPageCount(); }
}

```

```

void loadContent() override {
    if (hasAccess()) {
        realEBook->loadContent();
    } else {
        std::cout << "❌ دسترسی برای بارگذاری رد شد\n";
    }
}

bool isLoading() const override { return realEBook->isLoading(); }
};

// مثال Protection Proxy
void protectionProxyExample() {
    std::cout << "\n\n== مثال Protection Proxy ==\n";
    // ایجاد کاربران
    auto admin = std::make_shared<RealUser>("علی", "Admin");
    auto premiumUser = std::make_shared<RealUser>("مریم", "Premium");
    auto freeUser = std::make_shared<RealUser>("رضا", "Free");
    // ایجاد کتاب
    auto realBook = std::make_shared<RealEBook>("کتاب محرمانه"
, "نویسنده ناشناس", 200);
    // ایجاد Proxy های حفاظت شده
    auto bookForAdmin = std::make_shared<ProtectedEBook>(realBook, admin);
    auto bookForPremium = std::make_shared<ProtectedEBook>(realBook, premiumUser);
    auto bookForFree = std::make_shared<ProtectedEBook>(realBook, freeUser);
    // تست دسترسی
    std::cout << "\n1. تلاش مدیر برای دسترسی:\n";
    bookForAdmin->display();

    std::cout << "\n2. برای دسترسی Premium تلاش کاربر:\n";
    bookForPremium->display();

    std::cout << "\n3. برای دسترسی Free تلاش کاربر:\n";
    bookForFree->display();
}

```

مثال 3: Smart Proxy برای Logging و Caching 🍌

```

// با قابلیت های اضافه Smart Proxy
class SmartEBookProxy : public EBook {
private:
    std::shared_ptr<RealEBook> realEBook;
    mutable int accessCount = 0;
    mutable std::string cachedDisplay;
    mutable bool cacheValid = false;
    mutable std::mutex cacheMutex;

    void logAccess(const std::string& operation) const {
        std::cout << "[SmartProxy] " << operation
        << " - کتاب: " << realEBook->getTitle()
        << " - تعداد دسترسی: " << (accessCount + 1) << "\n";
    }

public:
    SmartEBookProxy(const std::shared_ptr<RealEBook>& ebook)
        : realEBook(ebook) {}

    void display() const override {
        std::lock_guard<std::mutex> lock(cacheMutex);
        accessCount++;
        logAccess("نمایش");
    }
}

```

```

        if (cacheValid && realEBook->isLoaded()) {
            std::cout << "[SmartProxy] استفاده از cache\n";
            std::cout << cachedDisplay;
        } else {
            realEBook->display();
            // کش کردن خروجی
            if (realEBook->isLoaded()) {
                std::stringstream ss;
                ss << "\n=== " << realEBook->getTitle() << " ===\n"
                    << "نویسنده: " << realEBook->getAuthor() << "\n"
                    << "تعداد صفحات: " << realEBook->getPageCount() << "\n"
                    << "cache این نمایش از "\n";
                cachedDisplay = ss.str();
                cacheValid = true;
            }
        }
        std::cout << "[SmartProxy] این کتاب " << accessCount
            << "\nبار مشاهده شده است " << "\n";
    }

    std::string getTitle() const override {
        logAccess("دریافت عنوان");
        return realEBook->getTitle();
    }

    std::string getAuthor() const override {
        logAccess("دریافت نویسنده");
        return realEBook->getAuthor();
    }

    int getPageCount() const override {
        logAccess("دریافت تعداد صفحات");
        return realEBook->getPageCount();
    }

    void loadContent() override {
        logAccess("بارگذاری");
        realEBook->loadContent();
        cacheValid = false; // کش نامعتبر شد
    }

    bool isLoaded() const override {
        return realEBook->isLoaded();
    }

    int getAccessCount() const { return accessCount; }

    void invalidateCache() {
        std::lock_guard<std::mutex> lock(cacheMutex);
        cacheValid = false;
        std::cout << "[SmartProxy] cache کتاب '"
            << realEBook->getTitle() << "' پاک شد.\n";
    }
};

// مثال Smart Proxy
void smartProxyExample() {
    std::cout << "\n\n=== مثال Smart Proxy ===\n";

    auto realBook = std::make_shared<RealEBook>(
        150, "کتاب هوشمند", "نویسنده هوشمند", 150);

```

```

SmartEBookProxy smartBook(realBook);

std::cout << "\n مشاهده بار اولین بار (بدون cache):\n";
smartBook.loadContent();
smartBook.display();

std::cout << "\n مشاهده بار دوم (با cache):\n";
smartBook.display();

std::cout << "\n مشاهده بار سوم (با cache):\n";
smartBook.display();

std::cout << "\n تعداد کل دسترسی‌ها: "
          << smartBook.getAccessCount() << "\n";
}

```

مثال ۴: Remote Proxy (شبیه‌سازی) 🍌

```

// Remote Proxy شبیه‌سازی
class RemoteEBookService {
public:
    // شبیه‌سازی درخواست شبکه
    std::string fetchFromServer(const std::string& bookId) {
        std::cout << "[RemoteService] اتصال به سرور...\n";
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << "[RemoteService] دریافت داده برای کتاب " << bookId <<
        "... \n";
        std::this_thread::sleep_for(std::chrono::seconds(2));
        return "ID: محتوای کتاب از سرور دور برای " + bookId;
    }
};

class RemoteEBookProxy : public EBook {
private:
    std::string bookId;
    std::string title;
    std::string author;
    int pageCount;
    mutable std::string cachedContent;
    mutable bool contentFetched = false;
    RemoteEBookService remoteService;

public:
    RemoteEBookProxy(const std::string& id, const std::string& title,
                     const std::string& author, int pages)
        : bookId(id), title(title), author(author), pageCount(pages) {}

    void display() const override {
        std::cout << "\n[RemoteProxy] نمایش کتاب '" << title << "'\n";

        loadContent();

        std::cout << "=== " << title << " ===\n"
                  << "نویسنده: " << author << "\n"
                  << "تعداد صفحات: " << pageCount << "\n"
                  << "محتوا: " << cachedContent << "\n";
    }

    std::string getTitle() const override { return title; }
    std::string getAuthor() const override { return author; }
    int getPageCount() const override { return pageCount; }
}

```

```

void loadContent() override {
    if (!contentFetched) {
        cachedContent = remoteService.fetchFromServer(bookId);
        contentFetched = true;
    }
}

bool isLoading() const override { return contentFetched; }
};
// مثال کامل با تمام Proxy ها
int main() {
    std::cout << "🌀 معماری نرم افزار در سی پلاس پلاس هادی عباسی\n\n";
    // 1. Virtual Proxy Example
    std::cout << "1. Virtual Proxy (Lazy Loading):\n";
    std::cout << "=====\n";

    DigitalLibrary library;
    // Proxy اضافه کردن کتابها با
    auto book1 = std::make_shared<EBookProxy>(100, "\", \"نویسنده \");
    auto book2 = std::make_shared<EBookProxy>(200, "\", \"نویسنده \");

    library.addEBook(book1);
    library.addEBook(book2);
    // نمایش بدون بارگذاری
    std::cout << "\nمرور کتابخانه:\n";
    library.browseLibrary();
    // بارگذاری و نمایش یک کتاب
    std::cout << "\nخواندن کتاب:\n";
    library.readEBook("\", \"کتاب\");

    // 2. Protection Proxy Example
    protectionProxyExample();
    // 3. Smart Proxy Example
    smartProxyExample();
    // 4. Remote Proxy Example
    std::cout << "\n\n=== مثال Remote Proxy ===\n";
    RemoteEBookProxy remoteBook("B123", 300, "\", \"نویسنده دور\");
    remoteBook.display();
    // (cache با استفاده از) نمایش دوباره
    std::cout << "\n(کش با) نمایش دوباره:\n";
    remoteBook.display();
    // 5. Smart + Virtual Proxy ترکیبی
    std::cout << "\n\n=== Smart Virtual Proxy ترکیبی ===\n";

    auto realBook = std::make_shared<RealEBook>("\", \"نویسنده \",
250);
    auto smartProxy = std::make_shared<SmartEBookProxy>(realBook);
    // اولین دسترسی
    std::cout << "\nاولین دسترسی:\n";
    smartProxy->display();
    // دسترسیهای بعدی
    std::cout << "\nدسترسیهای بعدی:\n";
    for (int i = 0; i < 3; i++) {
        smartProxy->display();
    }
    return 0;
}

```

✓ مزایای پترن Proxy

- کنترل دسترسی: مدیریت دسترسی به شیء اصلی
- Lazy Initialization: ایجاد شیء فقط هنگام نیاز
- Logging: ثبت فعالیت‌ها بدون تغییر کد اصلی
- Caching: بهبود عملکرد با ذخیره نتایج
- پیچیدگی پنهان: مخفی کردن پیچیدگی شیء اصلی

✗ معایب پترن Proxy

- پیچیدگی کد: اضافه کردن لایه‌های اضافی
- تأخیر پاسخ: ممکن است پاسخ‌دهی کندتر شود
- Maintenance: نیاز به نگهداری کد اضافی

🔧 کاربردهای واقعی

1- Lazy loading : Object Relational Mapping (ORM)

2- شبکه: Remote proxy برای RPC

3- مجوزدهی: Protection proxy برای کنترل دسترسی

4- Logging: ثبت خودکار فعالیت‌ها

5- Cache: ذخیره نتایج عملیات سنگین

🚀 نکات مهم پیاده‌سازی

1- Interface مشترک: Proxy و Real Subject باید اینترفیس یکسانی داشته باشند!

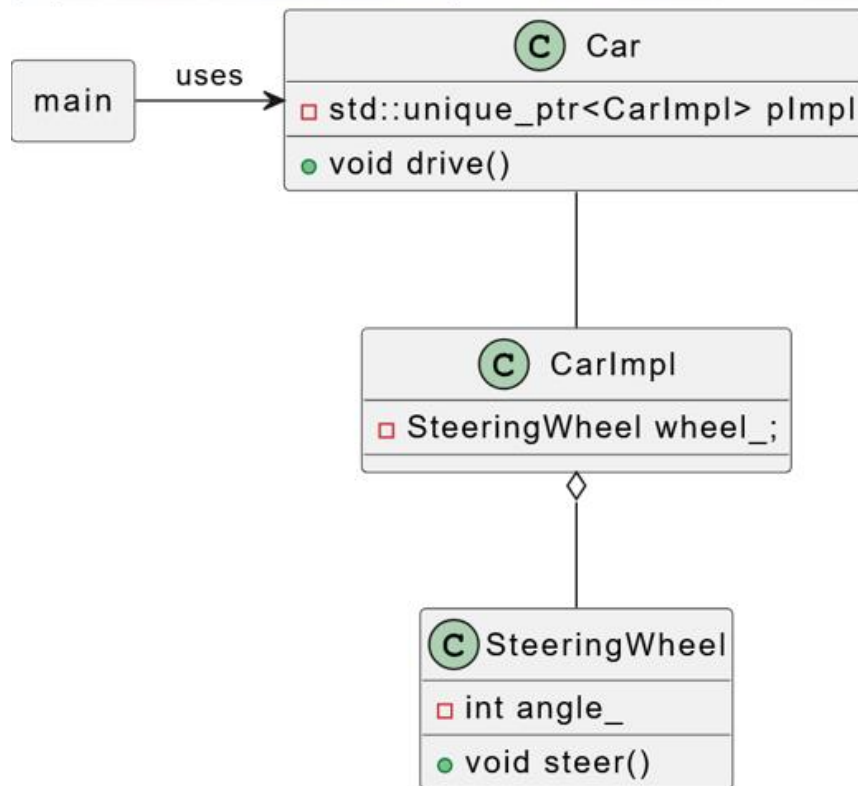
2- شفافیت: کلاینت نباید تفاوت بین Proxy و Real Subject را احساس کند!

3- مسئولیت‌های واضح: Proxy فقط باید مسئولیت‌های اضافه را انجام دهد و نهایتاً پس از احراز هویت و ... باید UseCase اصلی را به واسطه شیء اصلی اجرا نماید!

4- Thread Safety: در محیط‌های چندنخی احتیاط کنید!

این الگو زمانی مناسب است که نیاز به کنترل دسترسی، بهبود عملکرد یا اضافه کردن قابلیت‌های جانبی به یک شیء دارید بدون اینکه کد اصلی آن را تغییر دهید.

○ الگوی Pimple: یک C++ implementation idiom یا design idiom است و یک Design Pattern یا Architectural Pattern یا Enterprise Pattern نیست! ولی از نظر ساختاری به الگوهای structural نزدیک تر است، چون از ترکیب و encapsulation برای پنهان کردن پیاده‌سازی استفاده می‌کند. اما جزو الگوهای معروف GoF هم نیست!



تعریف ساده

الگوی PIMPL یعنی اینکه جزئیات پیاده‌سازی یک کلاس را در فایل `cpp` پنهان کنیم و در فایل هدر (`.h`) فقط یک اشاره‌گر (pointer) به آن پیاده‌سازی نگه داریم.

هدف اصلی:

- مخفی کردن جزئیات داخلی کلاس
- کاهش وابستگی‌ها
- کاهش زمان کامپایل
- پایدار نگه داشتن ABI

❌ مشکل بدون PIMPL چیست؟ فرض کنید در هدر کلاس شما:

```
// MyClass.h
#include <ClassA.h>
#include <ClassB.h>
Class MyClass {
public:
    MyClass() {}
private:
    ClassA ca;
    ClassB cb;
};
```

اگر کوچک‌ترین تغییری در `ClassA` یا `ClassB` بوجود آید، این کلاس هم دوباره کامپایل خواهد شد! اگر هر تغییری روی این کلاس از جمله روی یکی از اعضای خصوصی این کلاس دهید تمام فایل‌هایی که این هدر را `include` کرده‌اند، دوباره کامپایل می‌شوند و این در پروژه‌های بسیار بزرگ یعنی فاجعه `Build` کردن! به بیان بهتر اگر روی پروژه عظیمی در حال کار باشید، با هر تغییر کوچک روی این کلاسها باید تعداد بسیار زیادی از کلاسهای گاه حجیم دوباره کامپایل شوند! شاید حتی کامپایل‌های چند دقیقه‌ای یا شاید چند ساعته! این قضیه در جایی خسته کننده می‌شود که پس از مدت زمان قابل توجهی از `Build` کردن، ناگهان به چند خطای زمان `Compile` یا `Runtime` بخورید!

البته در `C++20` به بعد، سیستم `Module` جدیدی اضافه شده که با استفاده از ماژول نویسی هم می‌توانید این مساله را حل کنید! ولی من باب اهمیت `Pimple` در `C++`، در این بخش از کتاب به این موضوع می‌پردازیم!

ساختمان PIMP: در فایل هدر داریم:

```
// MyClass.h
#pragma once
#include <memory>

class MyClass {
public:
    MyClass();
    ~MyClass();
```



```

void doSomething();

private:
    class Impl; // forward declaration the class Impl
    std::unique_ptr<Impl> pImpl; // کلاس اصلی که مجری اصلی کاری خاص است
};

```

همان گونه که شاهد هستید، در هدر فقط یک **forward declaration** داریم و هیچ جزئیاتی مشخص نیست! اگر در هدر، به جای **Forward** کردن کلاس **Impl**، آنرا **include** کرده بودیم، مشکل اصلی آغاز می شد ولی اینجا از تکنیک **Forward Declaration** استفاده کردیم! دستور **Forward** کردن یک کلاس، یعنی به کامپایلر می گوییم، قطعا در آینده این کلاس را خواهی دید ولی فعلا پیشاپیش آنرا استفاده می کنم (به نوعی از آینده قرض می گیرم) تا در ادامه کامپایل، این کلاس را بیایی! البته شما نمی توانید در جایی که فوروارد کرده اید، به محتوای فایل مثل متدهای **public** و ... آن دسترسی پیدا کنید! ولی در کد زیر شاهد هستید که اصل فایل هدر کلاس **Impl** را داخل فایل **cpp**، اصطلاحا **include** کرده ایم و در اینجا می توانیم به خاطر **Include** کردن **Impl**، به تمام **property** ها و **method** های نوع **public** آن دسترسی پیدا کنیم و با آنها کار کنیم! از قضا مرحله بررسی فایل **cpp**، توسط کامپایلر، مرحله زمان **Link** شدن پس از کامپایل است و به همین جهت، با تغییر در فایل کلاس **Impl** دیگر **MyClass** و کلاسهایی که **MyClass** را **include** کرده اند دوباره کامپایل نخواهند شد! از طرفی فرض کنید به جای فایل **cpp**، از **.dll** یا **.so** به عنوان **Dynamic Lib** استفاده کنید و با این کار حتی می توانید جزئیات اصلی پیاده سازی کلاس **Impl** را به فرض محاسبات مهم و حساس، از دید کاربر مخفی نمایید!

```

// MyClass.cpp
#include "MyClass.h"
#include <vector>
#include <string>

class MyClass::Impl { // تعریف کامل کلاس ایمپل که در هدر به صورت فوروارد تعریف کردیم
    // کلاس ایمپل کلاس مجری اصلی یکی از عملیات اصلی کلاس بیرونی است
public:
    std::vector<int> data;
    std::string name;

    void doSomething() {
        // real logic here
    }
};

MyClass::MyClass() : pImpl(std::make_unique<Impl>()) {}

MyClass::~MyClass() = default;

void MyClass::doSomething() {
    pImpl->doSomething();
}

```

? چه چیزی مخفی شد؟

- تمام **include** ها
- تمام متغیرهای **private**
- ساختار داخلی کلاس
- الگوریتمها

کاربر کلاس فقط interface را می بیند. یادتان باشد که تغییر در فایل هدر، باعث کامپایل دوباره در تمام کلاس های استفاده کننده از کلاس تغییر یافته می شود!

مزایای PIMPL

- کاهش زمان کامپایل
- جداسازی Interface از Implementation
- مخفی سازی اطلاعات (Encapsulation قوی تر)
- حفظ ABI (خیلی مهم در کتابخانه های باینری)
- کاهش coupling

معایب PIMPL

- یک indirection اضافی (دسترسی از طریق pointer)
- مصرف حافظه بیشتر (یک pointer اضافه)
- نیاز به مدیریت صحیح Rule of Five
- پیچیدگی بیشتر در move/copy

چه زمانی استفاده می شود؟

- در پروژه های بزرگ C++
- در طراحی کتابخانه های عمومی
- در معماری های Clean/Layered
- در سیستم هایی که ABI stability مهم است (مثل Qt)

 **یک نگاه معماری:** PIMPL فقط یک تکنیک C++ نیست! در معماری نرم افزار، این الگو همان ایده ی Interface vs Implementation separation را در سطح کلاس پیاده سازی می کند. از نظر مفهومی بسیار نزدیک به Dependency Inversion Principle است و Encapsulation بهتر و مخفی سازی اطلاعات پیاده سازی (Information Hiding) را باعث می شود!

چالش های دیگری Pimple که باید حل شوند:

اگر کلاس شما به واسطه Copy Constructor یا Copy Assignment قابل copy شدن باشد، باید Copy Constructor و Copy Assignment را به درستی مدیریت کنید. چون std::unique_ptr به صورت پیش فرض copy نمی شود. پس برای copy کردن، باید Impl را عمیقاً کپی کنید. (Deep Copy) و اگر این کار را انجام ندهید، در کپی کردن شی ساخته شده از کلاس، Shallow Copy انجام می شود و شی اصلی Impl به صورت مشترک مورد استفاده هر دو یا چند کلاس کپی شده واقع می شود! همچنین نیاز به تعریف destructor در جایی که Impl کامل است داریم! به بیان بهتر نمی توانیم destructor پیش فرض یا default را به شکل زیر صدا بزنیم چون کلاس ما به جهت Forward Class از نوع InComplete است!

```
~Widget() = default; //Error: incomplete type
```

```
//widget.h
class Widget {
public:
    Widget();
    ~Widget();

    Widget(const Widget& other); //copy constructor
    Widget& operator=(const Widget& other); //copy assignment

    Widget(Widget&& other) noexcept; //move constructor
    Widget& operator=(Widget&& other) noexcept; //move assignment

private:
    class Impl;
    std::unique_ptr<Impl> pImpl;
};
```

```
#include "widget.h"
#include <iostream>
#include <string>

class Widget::Impl {
public:
    std::string name = "default";

    void draw() {
        std::cout << "Drawing: " << name << "\n";
    }
};

Widget::Widget()
    : pImpl(std::make_unique<Impl>()) {}

Widget::~~Widget() = default;

Widget::Widget(const Widget& other)
    : pImpl(std::make_unique<Impl>(*other.pImpl)) {}

Widget& Widget::operator=(const Widget& other) {
    if (this != &other) {
        *pImpl = *other.pImpl;
    }
    return *this;
}

Widget::Widget(Widget&& other) noexcept = default;

Widget& Widget::operator=(Widget&& other) noexcept = default;

void Widget::setName(const std::string& name) {
    pImpl->name = name;
}

void Widget::draw() {
    pImpl->draw();
}
```

فصل سوم

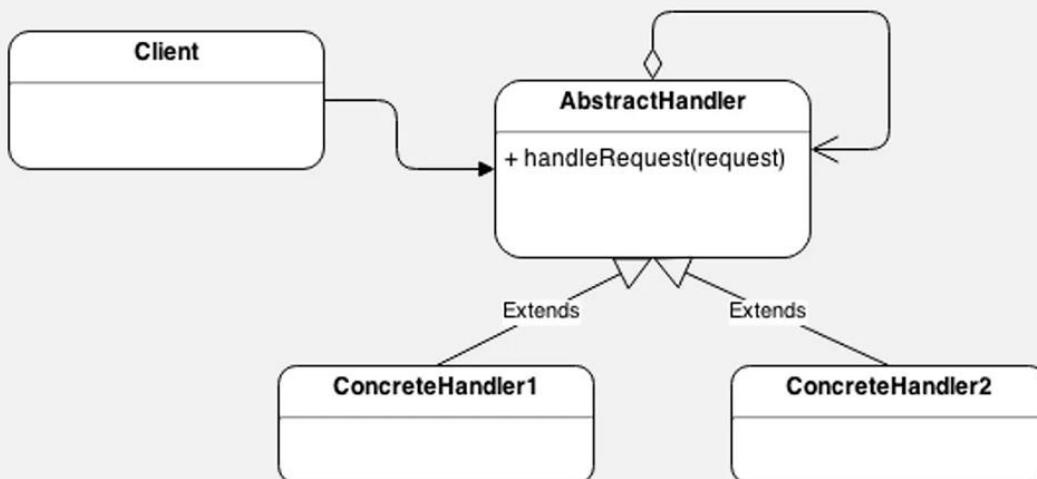
دیزاین پترنهای رفتاری (Behavioral)

این دسته از دیزاین پترن‌ها، به رفتار و ارتباط بین اشیا و تخصیص مسئولیت‌ها بین اشیا توجه می‌کنند. دیزاین پترنهای رفتاری، الگوهایی را توصیف می‌کنند که نه فقط الگوهای اشیا یا کلاس‌ها، بلکه الگوهای ارتباط بین آنها را نشان می‌دهند. این الگوها، جریان کنترل پیچیده‌ای را مشخص می‌کنند که در زمان اجرا دشوار است تا دنبال شود. انواع دیزاین پترنهای رفتاری عبارتند از:

○ **Chain of Responsibility**: پترنی از نوع Behavioral است!



Chain Of Responsibility pattern – Class diagram



این پترن یکی از پترن‌های رفتاری (Behavioral) است که درخواست را در زنجیره‌ای از اشیا انتقال می‌دهد و برای کاهش وابستگی بین فرستنده یک درخواست و گیرنده‌ی آن به کار می‌رود. در این پترن، چندین شی می‌توانند یک درخواست را پردازش کنند، اما فرستنده نمی‌داند کدام یک از آن‌ها این کار را انجام می‌دهد! تنها درخواست را به ابتدای یک «زنجیره» می‌فرستد و هر شی در زنجیره تصمیم می‌گیرد که آیا آن را پردازش کند یا به شی بعدی در زنجیره منتقل کند.

به بیان بهتر فرض کنید وارد سرویس کاری برای خودرویتان می‌شوید و از ابتدای زنجیره، تک تک بخش‌های خودرو را بررسی می‌کند و اگر هر بخش مشکلی داشته باشد، آنرا رفع می‌کند و سپس شما را به سرویس کار بخش بعدی می‌فرستد تا اینکه پس از پایان کامل زنجیره سرویس، خودرویتان به صورت کامل سرویس و تعمیر شده باشد!

ایده‌ی کلی: هر عنصر در زنجیره :

○ یا درخواست را پردازش می‌کند،

○ یا آن را به عنصر بعدی در زنجیره هدایت می‌کند.

این کار انعطاف‌پذیری بیشتری در پردازش درخواست‌ها فراهم می‌کند، خصوصاً وقتی نوع درخواست از پیش مشخص نیست یا منطق پردازش متغیر است.

ساختار در C++ : برای این کار در C++ معمولاً :

1. یک کلاس پایه‌ی انتزاعی (Handler) داریم که یک رابط مشترک برای تمام هندلرها تعریف می‌کند.

2. هر هندلر اشاره‌گری به هندلر بعدی دارد.

3. درخواست از ابتدای زنجیره شروع می‌شود.

مثال کاربردی: سیستم لاگ‌گیری با سطوح مختلف

فرض کنید می‌خواهیم یک سیستم لاگ‌گیری بسازیم که سطوح زیر را پشتیبانی کند :

- INFO
- WARNING
- ERROR

هر هندلر فقط مواردی را لاگ می‌کند که مربوط به سطح خودش یا بالاتر از آن باشد :

- InfoLogger: فقط INFO
- WarningLogger: WARNING و INFO
- ErrorLogger: همه چیز (ERROR, WARNING, INFO)

اما ما می‌خواهیم درخواست را فقط یک‌بار بفرستیم و بگذاریم زنجیره تصمیم بگیرد .

```
#include <iostream>
#include <memory>
#include <string>
enum class LogLevel {
    INFO,
    WARNING,
    ERROR
};
```

```

class Logger { // Handler پایه
protected:
    std::unique_ptr<Logger> nextLogger;

public:
    virtual ~Logger() = default;

    void setNext(std::unique_ptr<Logger> next) {
        nextLogger = std::move(next);
    }

    void log(LogLevel level, const std::string& message) {
        if (canHandle(level)) {
            writeMessage(message);
        }
        if (nextLogger) {
            nextLogger->log(level, message);
        }
    }
protected:
    virtual bool canHandle(LogLevel level) = 0;
    virtual void writeMessage(const std::string& message) = 0;
};

// هندلرهای خاص
class InfoLogger : public Logger {
protected:
    bool canHandle(LogLevel level) override {
        return level == LogLevel::INFO;
    }

    void writeMessage(const std::string& message) override {
        std::cout << "[INFO] " << message << std::endl;
    }
};

class WarningLogger : public Logger {
protected:
    bool canHandle(LogLevel level) override {
        return level == LogLevel::WARNING;
    }

    void writeMessage(const std::string& message) override {
        std::cout << "[WARNING] " << message << std::endl;
    }
};

class ErrorLogger : public Logger {
protected:
    bool canHandle(LogLevel level) override {
        return level == LogLevel::ERROR;
    }

    void writeMessage(const std::string& message) override {
        std::cout << "[ERROR] " << message << std::endl;
    }
};

```

```
// استفاده
int main() {
    auto errorLogger = std::make_unique<ErrorLogger>();
    auto warningLogger = std::make_unique<WarningLogger>();
    auto infoLogger = std::make_unique<InfoLogger>();

    // ساخت زنجیره: Error -> Warning -> Info
    errorLogger->setNext(std::move(warningLogger));
    errorLogger->nextLogger->setNext(std::move(infoLogger));
    Logger* chain = errorLogger.get(); // دسترسی به شی خام اصلی
    chain->log(LogLevel::INFO, "This is an info message");
    chain->log(LogLevel::WARNING, "This is a warning message");
    chain->log(LogLevel::ERROR, "This is an error message");

    return 0;
}
```

خروجی: 

```
[INFO] This is an info message
[WARNING] This is a warning message
[ERROR] This is an error message
```

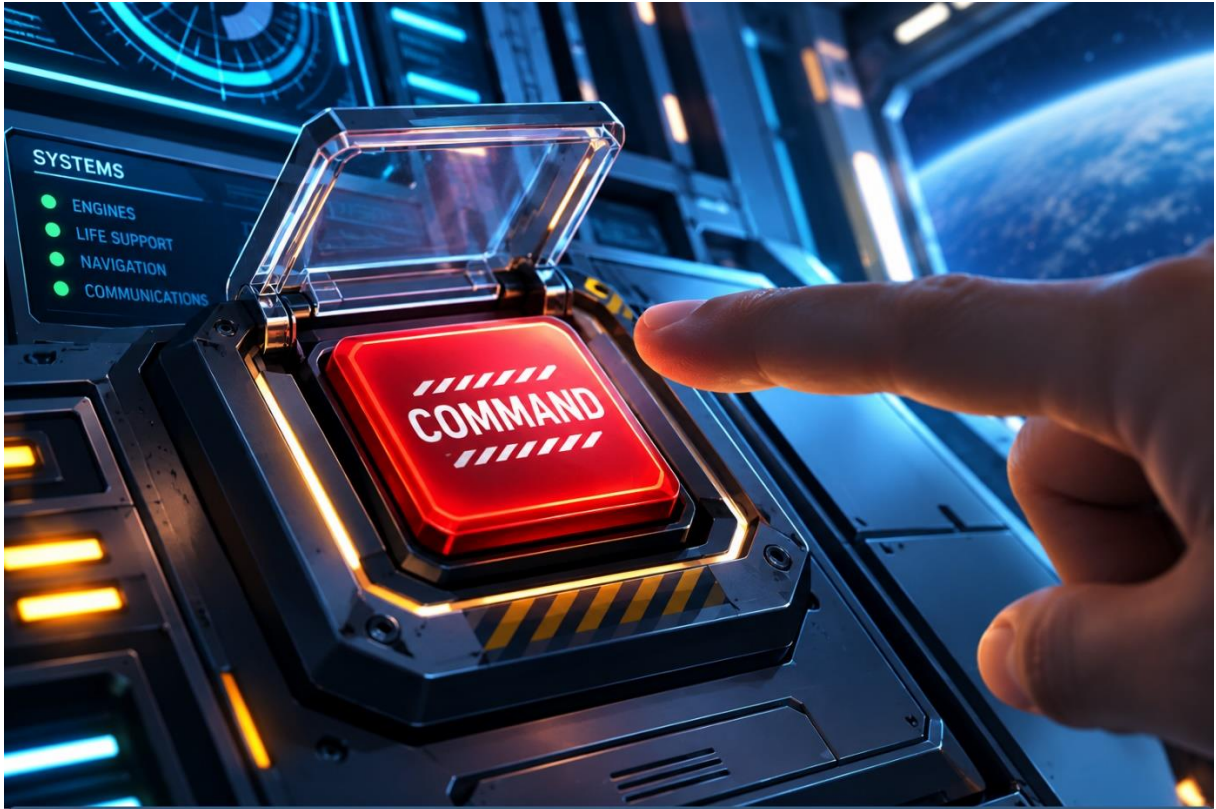
 **توجه:** در این مثال، فقط یک هندلر هر بار درخواست را پردازش می‌کند. اگر می‌خواستیم همه‌ی هندلرهایی که مجازند پردازش کنند (مثل اینکه ERROR هم در ERROR Logger و هم در Warning و Info چاپ شود)، باید منطق `canHandle` را تغییر می‌دادیم! همچنان که شاهد هستید، با این روش، اگر هر نوع Log ارسال کنیم، توسط مسئول مربوطه اش اجرا و لاگ خواهد شد!

مزایای Chain of Responsibility

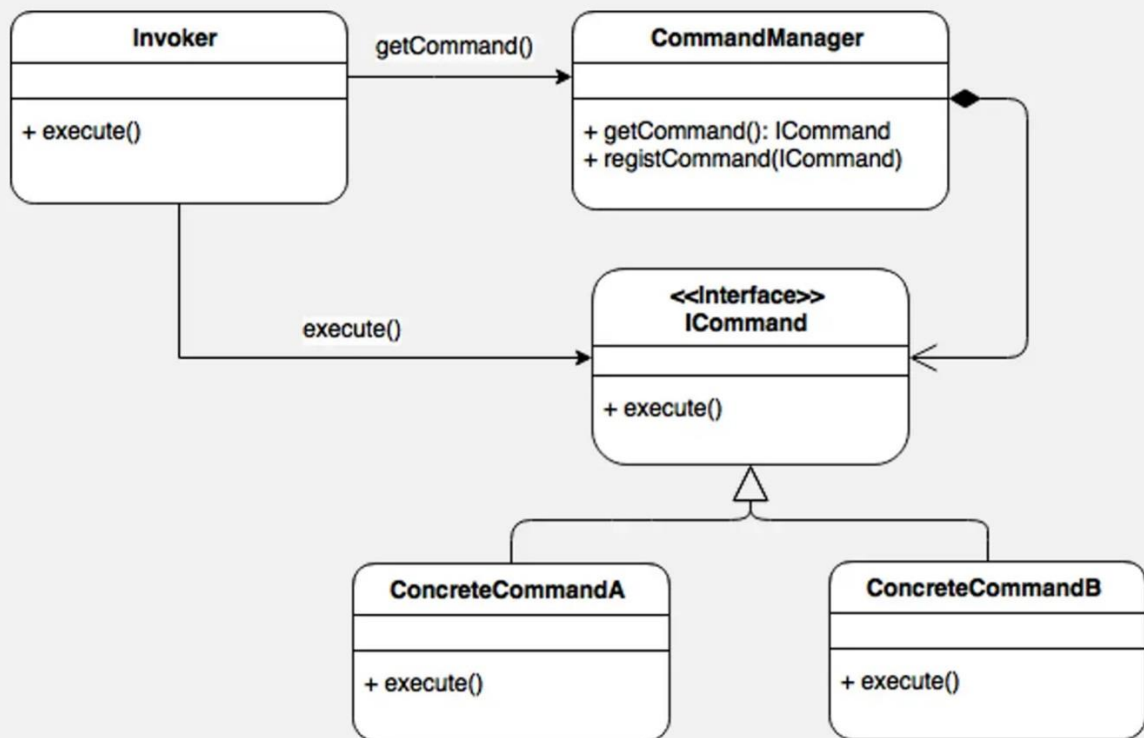
- کاهش جفت‌شدگی: **(Decoupling)**: فرستنده نیازی به دانستن جزئیات پردازنده ندارد.
- انعطاف‌پذیری در ساخت زنجیره: می‌توان زنجیره را در زمان اجرا تغییر داد.
- پیاده‌سازی منطق شرطی پیچیده به صورت ماژولار.

کاربردهای واقعی

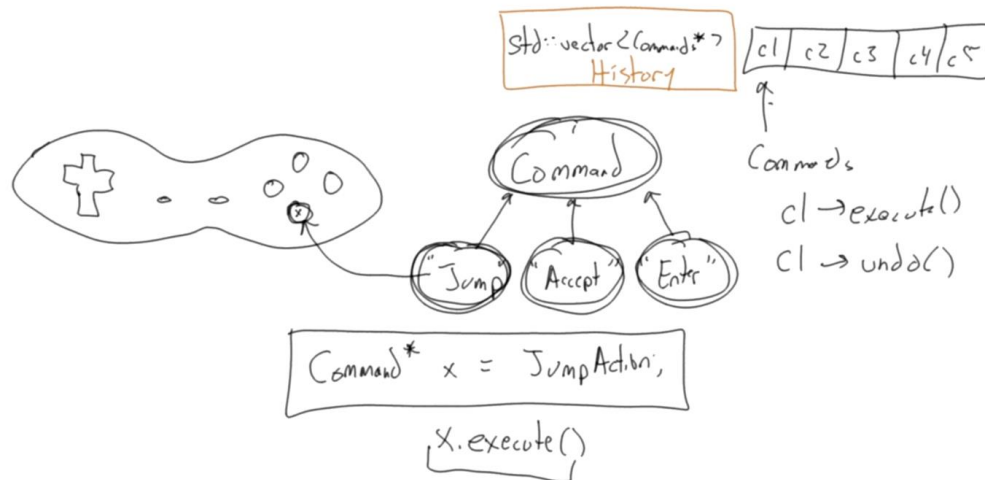
- سیستم‌های لاگ‌گیری با سطوح مختلف.
- پردازش درخواست‌های HTTP در Middleware مثل Express.js یا ASP.NET Core.
- مدیریت Event در UI، مثلاً اگر یک کلیک در یک کنترل پردازش نشود، به پدر منتقل شود. (استفاده در الگوی Composite)
- احراز هویت و دسترسی در سرویس‌های مایکروسروسی (هر سرویس یک بخش از اعتبارسنجی را بررسی می‌کند).



Command pattern – Class diagram



این دیزاین پترن در نوشتن اپلیکیشن هایی که خاصیت Undo-Redo دارند بسیار پرکاربرد است. هر دکمه یا هر اتفاق نهایی، مثل جا به جایی یک شی از نقطه A به B توسط درگ اند دراپ و ... یک command تلقی می شود. ما برای command ها یک کلاس پایه تعریف می کنیم. این کلاس پایه متدهای execute و undo (و می توان Redo هم تعریف کرد) و Destructor از نوع abstract یا pure virtual دارد. سپس کلاسهای مشتق شده از آن کلاس را برای subclass هایی همچون Move یا Jump یا Fire تعریف می کنیم که این متدهای abstract را Override می کنند. یک stack هم از نوع اشاره گر Command داریم که هر دستور در آن push می شود.



کد استفاده از این الگوی طراحی در C++ به این صورت است: 🔑

```
#include <iostream>
#include <string>
#include <stack>
// The Character class represents the receiver of the commands
class Character {
private:
    std::string name;
    int x;
    int y;
public:
    // Constructor
    Character(std::string n, int x0, int y0) : name(n), x(x0), y(y0) {}
    // Getter methods
    std::string getName() { return name; }
    int getX() { return x; }
    int getY() { return y; }
    // The move method performs the action of changing the position of the character
    void move(int newX, int newY) {
        std::cout << name << " moved from (" << x << ", " << y << ") to (" << newX << ", " << newY << ")\n";
        x = newX;
        y = newY;
    }
};
// The Command interface declares the common methods for all commands
class Command {
public:
    virtual void execute() = 0; // Execute the command
    virtual void undo() = 0; // Undo the command
    virtual ~Command() {} // Destructor
};
```

```

// The MoveCommand class is a concrete command that implements the Command interface
class MoveCommand : public Command {
private:
    Character& character; // The reference to the receiver
    int lastX; // The previous x-coordinate of the character
    int lastY; // The previous y-coordinate of the character
    int newX; // The new x-coordinate of the character
    int newY; // The new y-coordinate of the character

public:
    // Constructor
    MoveCommand(Character& c, int x, int y) : character(c), newX(x), newY(y) {
        // Save the current position of the character
        lastX = character.getX();
        lastY = character.getY();
    }
    // The execute method calls the move method on the receiver with the new coordinates
    void execute() {
        character.move(newX, newY);
    }
    // The undo method calls the move method on the receiver with the previous coordinates
    void undo() {
        character.move(lastX, lastY);
    }
};

// The Invoker class is responsible for initiating the execution of commands
class Invoker {
private:
    std::stack<Command*> commands; //A stack of commands to store the history of actions
public:
    // The invoke method executes a command and pushes it to the stack
    void invoke(Command* command) {
        command->execute();
        commands.push(command);
    }
    // The undo method pops a command from the stack and undoes it
    void undo() {
        if (!commands.empty()) {
            Command* command = commands.top();
            command->undo();
            commands.pop();
            delete command; // Free the memory
        }
        else {
            std::cout << "No commands to undo\n";
        }
    }
};

int main() {
    // Create a character object
    Character character("Alice", 0, 0);
    // Create an invoker object
    Invoker invoker;
    // Create and invoke some move commands
    invoker.invoke(new MoveCommand(character, 1, 2));
    invoker.invoke(new MoveCommand(character, 3, 4));
    invoker.invoke(new MoveCommand(character, 5, 6));
    // Undo some commands
    invoker.undo();
    invoker.undo();
}

```

```

invoker.undo();
invoker.undo();

return 0;
}

```

نتیجه این کد: 

```

Alice moved from (0, 0) to (1, 2)
Alice moved from (1, 2) to (3, 4)
Alice moved from (3, 4) to (5, 6)
Alice moved from (5, 6) to (3, 4)
Alice moved from (3, 4) to (1, 2)
Alice moved from (1, 2) to (0, 0)
No commands to undo

```

 **نکته مهم:** ما دستورات اصلی این برنامه مثل حرکت یا شلیک کاراکتر یا جا به جایی یک شی از نقطه A به نقطه B در حین درگ اند دراپ را با ساخت و `execute` هر `command` مرتبط اجرا می کنیم و لذا به صورت مستقل بدون هماهنگی با مدیریت کننده `command` ها انجام نمی دهیم!

 **نکته مهم:** اگر بخواهیم مکانیسم `redo` هم داشته باشیم، از 2 پشته یکی برای `undo` و دیگری برای `Redo` استفاده می کنیم. در این مثال می خواهیم سایی هم برای تعداد `undo-redo` ها تنظیم کنیم. نکته مهمی که وجود دارد اینست که با `Undo` کردن، آیتم جدید از پشته `Undo` به پشته `Redo` منتقل می شود و با هر `Redo` کردن، آیتم بالای `Redo` پس از `execute` شدن وارد پشته `Undo` می شود! در قسمت بعد کد مربوط به `Undo` کردن به همراه `Redo` کردن را ارائه می دهیم!

```

class Invoker {
private:
    std::stack<Command*> undoStack; // A stack of commands to store the history of
    actions
    std::stack<Command*> redoStack; // A stack of commands to store the history of
    undone actions
    const int MAX_SIZE = 10; // The maximum size of the stacks
public:
    // The invoke method executes a command and pushes it to the undo stack
    // It also clears the redo stack
    void invoke(Command* command) {
        command->execute();
        undoStack.push(command);
        // If the undo stack exceeds the maximum size, discard the oldest command
        if (undoStack.size() > MAX_SIZE) {
            std::stack<Command*> tempStack;
            while (!undoStack.empty())
            {
                tempStack.push(undoStack.top());
                undoStack.pop();
            }

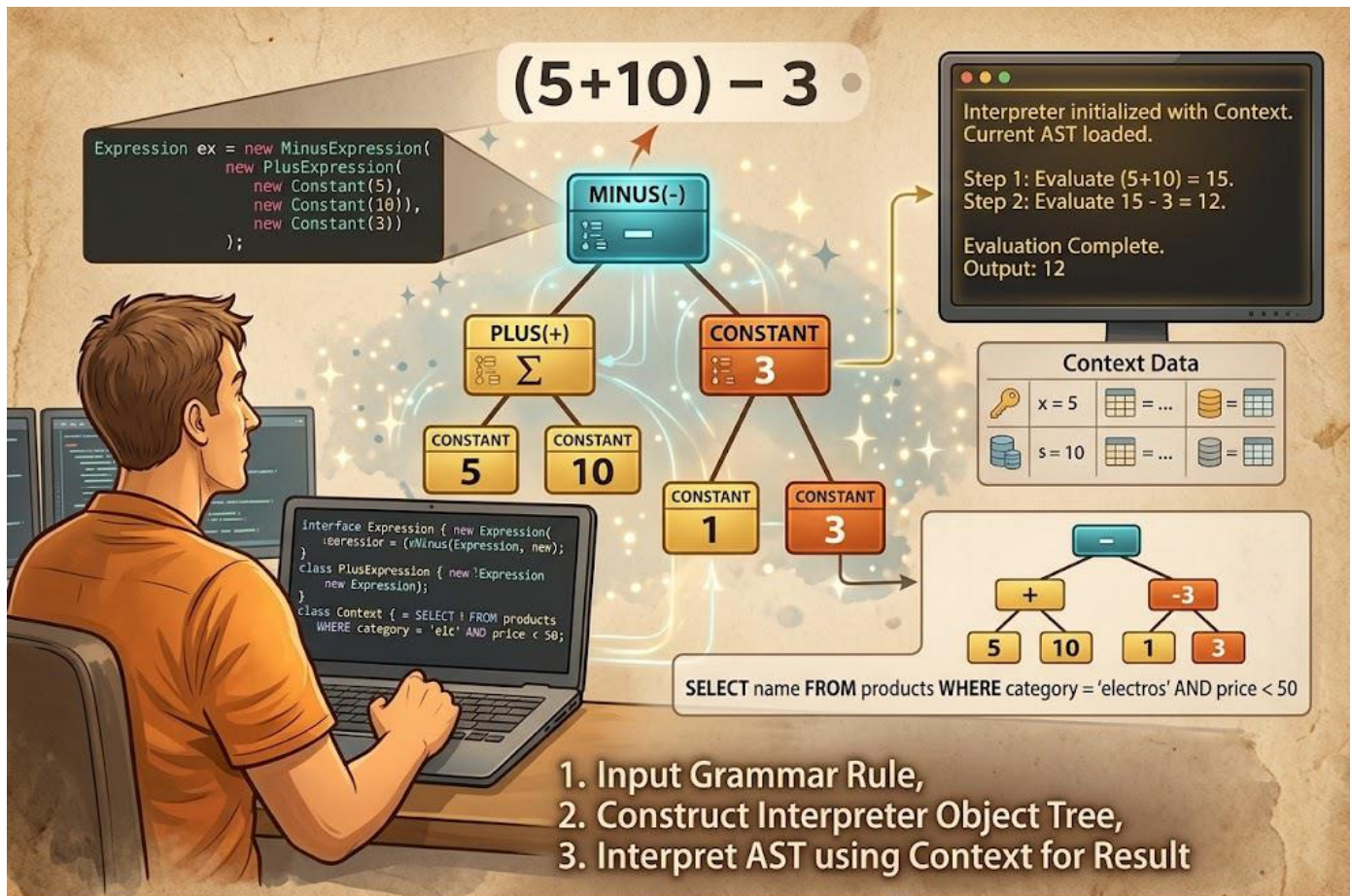
```

```

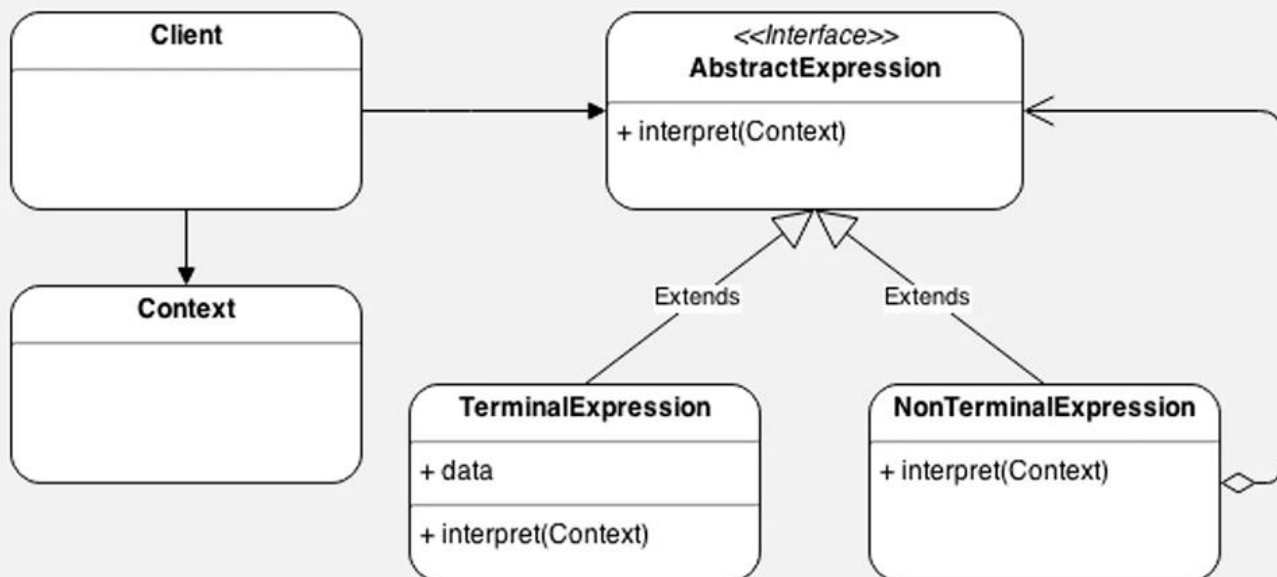
        // The first added item is now on top of the temporary stack.
        Command* oldItem = tempStack.top();
        tempStack.pop();
        delete oldItem;
        // If you need to pop the first added item:
        // Transfer elements back to the original stack if needed.
        while (!tempStack.empty())
        {
            undoStack.push(tempStack.top());
            tempStack.pop();
        }
    }
    // Clear the redo stack
    while (!redoStack.empty()) {
        Command* oldCommand = redoStack.top();
        redoStack.pop();
        delete oldCommand; // Free the memory
    }
}
// The undo method pops a command from the undo stack and undoes it
// It also pushes it to the redo stack
void undo() {
    if (!undoStack.empty()) {
        Command* command = undoStack.top();
        command->undo();
        undoStack.pop();
        redoStack.push(command);
    }
    else {
        std::cout << "No commands to undo\n";
    }
}
// The redo method pops a command from the redo stack and executes it
// It also pushes it to the undo stack
void redo() {
    if (!redoStack.empty()) {
        Command* command = redoStack.top();
        command->execute();
        redoStack.pop();
        undoStack.push(command);
    }
    else {
        std::cout << "No commands to redo\n";
    }
}
};

```

Interpreter: پترنی از نوع Behavioral است و دستورات زبان خاصی را تفسیر می کند!



Interpreter pattern – Class diagram



🏠 پترن Interpreter چیست؟

Interpreter Pattern زمانی استفاده می‌شود که:

- یک زبان ساده یا DSL داریم!
- می‌خواهیم عبارات (Expressions) آن زبان را تفسیر کنیم!
- گرامر زبان نسبتاً ساده است!
- می‌خواهیم بتوانیم قوانین را توسعه بدهیم، آن هم بدون دست زدن به کل سیستم!

🏠 تعریف Gof:

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

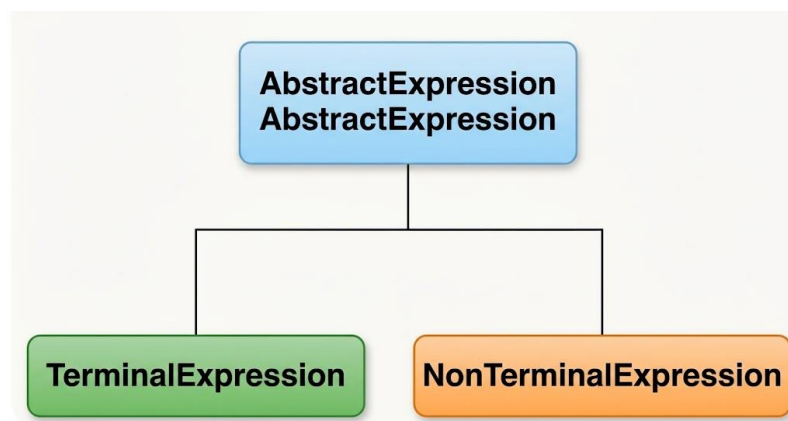
🔑 موارد کاربرد واقعی:

- Query های ساده (SQL-like)
- Rule Engine
- Expression Evaluator
- Validation rules
- فیلترها (Search / Permissions)
- محاسبات مالی یا منطقی

❌ چه زمانی نباید استفاده شود؟

- وقتی grammar خیلی پیچیده است!
- وقتی performance بسیار حیاتی است!
- وقتی می‌توان از Parser Generator هایی مثل ANTLR استفاده کرد!

🏠 ساختار کلی پترن Interpreter:



1. **AbstractExpression** → اینترفیس تفسیر
2. **TerminalExpression** → عناصر پایه زبان یا Operand ها یا Variable ها (عملوندها)
3. **NonTerminalExpression** → ترکیب عبارات یا Operator ها (عملگرها)
4. **Context** → اطلاعات اجرایی (متغیرها، محیط اجرا)

در این مبحث Context حکم مکان نگه داری متغیرها همراه با مقادیرشان را دارد! به بیان بهتر یک map شامل کلید نام متغیرها از نوع رشته و value کلیدها از نوع Int (می توان از اعداد اعشاری هم استفاده کرد!) TerminalExpression ها نماد اشیای عملوند هستند که مقادیرشان را از Context می گیرند و NonTerminalExpression ها حکم عملگرها را دارند که دو عملوند در چپ و راستشان قرار می گیرند و نتیجه مدنظر خود را به بیرون می دهند! مثال پایه (ریاضی ساده):

a + b - c

تعریف Context

```
#include <string>
#include <unordered_map>
class Context {
public:
    void setVariable(const std::string& name, int value) {
        variables[name] = value;
    }
    int getVariable(const std::string& name) const {
        return variables.at(name);
    }

private:
    std::unordered_map<std::string, int> variables;
};
```

Abstract Expression

```
class Expression {
public:
    virtual ~Expression() = default;
    virtual int interpret(const Context& context) const = 0;
};
```

Terminal Expression (متغیر)

```
class VariableExpression : public Expression {
public:
    explicit VariableExpression(std::string name)
        : name_(std::move(name)) {}

    int interpret(const Context& context) const override {
        return context.getVariable(name_);
    }

private:
    std::string name_;
};
```

Non Terminal Expression (عملگر جمع):

```
#include <memory>
class AddExpression : public Expression {
public:
    AddExpression(std::unique_ptr<Expression> left,
                  std::unique_ptr<Expression> right)
        : left_(std::move(left)), right_(std::move(right)) {}

    int interpret(const Context& context) const override {
        return left_>interpret(context) + right_>interpret(context);
    }
private:
    std::unique_ptr<Expression> left_;
    std::unique_ptr<Expression> right_;
};
```

Non Terminal Expression (تفریق):

```
class SubtractExpression : public Expression {
public:
    SubtractExpression(std::unique_ptr<Expression> left,
                      std::unique_ptr<Expression> right)
        : left_(std::move(left)), right_(std::move(right)) {}

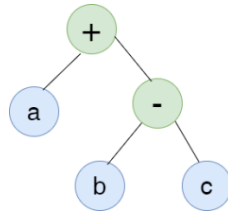
    int interpret(const Context& context) const override {
        return left_>interpret(context) - right_>interpret(context);
    }
private:
    std::unique_ptr<Expression> left_;
    std::unique_ptr<Expression> right_;
};
```

استفاده نهایی: 

```
int main() {
    Context context;
    context.setVariable("a", 10);
    context.setVariable("b", 5);
    context.setVariable("c", 3);
    // a + b - c
    auto expression =
        std::make_unique<SubtractExpression>(
            std::make_unique<AddExpression>(
                std::make_unique<VariableExpression>("a"),
                std::make_unique<VariableExpression>("b")
            ),
            std::make_unique<VariableExpression>("c")
        );
    std::cout << expression->interpret(context) << std::endl; // 12
}
```

همانطور که می دانید، یک عبارت محاسباتی پیچیده حاصل ترکیب چند عبارت ساده محاسباتی است که بر اساس علامات، اولویت بندی می شوند و در ساده ترین جزئیات، مقدار داخل متغیرها یا عملوند ها (ترمینال ها) وارد محاسبه می شوند و عملگر بر اساس مقادیر آنها، محاسبات را انجام داده و نتیجه محاسبات با اولویت بالا، وارد عملوند محاسبات با اولویت پایین تر می شوند! مثلاً در مثال زیر:

$a + (b - c)$



در عبارت بالا در هر صورت اول باید $b - c$ محاسبه شود و نتیجه این محاسبه وارد محاسبه Operator بعدی یعنی $+$ شود!

🧠 مثال حرفه‌ای تر Rule Engine (بسیار کاربردی شبیه به دستورات SQL)

`(age > 18) AND (country == "IR")`

Context پیشرفته: 🚀

```
#include <variant>

using Value = std::variant<int, std::string, bool>;

class Context {
public:
    void set(const std::string& key, Value value) {
        data[key] = value;
    }

    const Value& get(const std::string& key) const {
        return data.at(key);
    }
private:
    std::unordered_map<std::string, Value> data;
};
```

AbstractExpression (Boolean)

```
class BoolExpression {
public:
    virtual ~BoolExpression() = default;
    virtual bool interpret(const Context& ctx) const = 0;
};
```

Terminal: Comparison

```
class EqualsExpression : public BoolExpression {
public:
    EqualsExpression(std::string key, Value expected)
        : key_(std::move(key)), expected_(std::move(expected)) {}
    bool interpret(const Context& ctx) const override {
        return ctx.get(key_) == expected_;
    }
private:
    std::string key_;
    Value expected_;
};
```

```

class GreaterThanExpression : public BoolExpression {
public:
    GreaterThanExpression (std::string key, Value expected)
        : key_(std::move(key)), expected_(std::move(expected)) {}

    bool interpret(const Context& ctx) const override {
        return ctx.get(key_) > expected_;
    }
private:
    std::string key_;
    Value expected_;
};

```

NonTerminal: AND (برای Terminal بودن باید اشاره گر های Left و Right را داشته باشد!)

```

class AndExpression : public BoolExpression {
public:
    AndExpression(std::unique_ptr<BoolExpression> left,
                  std::unique_ptr<BoolExpression> right)
        : left_(std::move(left)), right_(std::move(right)) {}

    bool interpret(const Context& ctx) const override {
        return left_->interpret(ctx) && right_->interpret(ctx);
    }
private:
    std::unique_ptr<BoolExpression> left_;
    std::unique_ptr<BoolExpression> right_;
};

```

استفاده 

```

// (country == "IR") AND (age == 20)
Context ctx;
ctx.set("age", 20);
ctx.set("country", std::string("IR"));

auto rule = std::make_unique<AndExpression>(
    std::make_unique<EqualsExpression>("country", std::string("IR")),
    std::make_unique<EqualsExpression>("age", 20)
);

bool allowed = rule->interpret(ctx); //true
// (country == "IR") AND (age > 18)
auto rule2 = std::make_unique<AndExpression>(
    std::make_unique<EqualsExpression>("country", std::string("IR")),
    std::make_unique<GreaterThanExpression>("age", 18)
);

bool allowed2 = rule2->interpret(ctx); //true

```

مزایا

- توسعه پذیری بالا
- خوانایی بالا
- **Open/Closed Principle**
- مناسب برای DSL
- سادگی تغییر و گسترش دستور زبان: اضافه کردن قوانین جدید آسان است!
- پیاده سازی ساده: هر قانون در یک کلاس جداگانه پیاده سازی می شود!
- قابلیت افزودن راه های جدید تفسیر: می توان عملیات جدیدی به مفسر اضافه کرد!

معایب

- **Performance** پایین تر: ممکن است برای دستورات پیچیده کارایی مطلوبی نداشته باشد!
- نامناسب برای **grammar** پیچیده: برای دستور زبان های پیچیده، تعداد کلاس ها زیاد می شود!
- سختی نگهداری: با بزرگ شدن دستور زبان، نگهداری سخت می شود!

نوشتن یک Expression Parser ساده که بر اساس رشته های ورودی، Expression ها را بسازد: 

```
#include <iostream>
#include <string>
#include <vector>
#include <stack>
#include <sstream>
#include <cctype>
#include <memory>
#include <stdexcept>

// =====
// بخش ۱: کلاسهای الگوی Interpreter برای ساخت درخت عبارت
// =====

class Expression {
public:
    virtual ~Expression() = default;
    virtual double Interpret() = 0;
    virtual std::string ToString() const = 0;
};

class NumberExpression : public Expression {
    double value;
public:
    explicit NumberExpression(double v) : value(v) {}
    double Interpret() override { return value; }
    std::string ToString() const override { return std::to_string(value); }
};
```

```

class AddExpression : public Expression {
    std::unique_ptr<Expression> _left, _right;
public:
    AddExpression(std::unique_ptr<Expression> left,
                  std::unique_ptr<Expression> right) :
        _left(std::move(left)),
        _right(std::move(right)) {}
    double Interpret() override { return _left->Interpret() + _right->Interpret(); }
    std::string ToString() const override { return "(" + _left->ToString() + " + "
+ _right->ToString() + ")"; }
};

class SubtractExpression : public Expression {
    std::unique_ptr<Expression> _left, _right;
public:
    SubtractExpression(std::unique_ptr<Expression> left,
                      std::unique_ptr<Expression> right) :
        _left(std::move(left)),
        _right(std::move(right)) {}
    double Interpret() override { return _left->Interpret() - _right->Interpret(); }
    std::string ToString() const override { return "(" + _left->ToString() + " - "
+ _right->ToString() + ")"; }
};

class MultiplyExpression : public Expression {
    std::unique_ptr<Expression> _left, _right;
public:
    MultiplyExpression(std::unique_ptr<Expression> left,
                      std::unique_ptr<Expression> right) :
        _left(std::move(left)),
        _right(std::move(right)) {}
    double Interpret() override { return _left->Interpret() * _right->Interpret(); }
    std::string ToString() const override { return "(" + _left->ToString() + " * "
+ _right->ToString() + ")"; }
};

class DivideExpression : public Expression {
    std::unique_ptr<Expression> _left, _right;
public:
    DivideExpression(std::unique_ptr<Expression> left,
                    std::unique_ptr<Expression> right) :
        _left(std::move(left)),
        _right(std::move(right)) {}
    double Interpret() override {
        double rVal = _right->Interpret();
        if (rVal == 0) throw std::runtime_error("Division by zero!");
        return _left->Interpret() / rVal;
    }
    std::string ToString() const override { return "(" + _left->ToString() + " / "
+ _right->ToString() + ")"; }
};

// =====
// بخش ۲: تبدیل عبارت ریاضی (Infix) به پسوندی (Postfix)
// =====
std::string infixToPostfix(const std::string& infix) {
    // حذف فاصله‌های اضافی
    std::string cleaned;
    for (char c : infix) {
        if (!std::isspace(static_cast<unsigned char>(c))) cleaned += c;
    }
}

```

```

// توکن بندی (جدا کردن اعداد از عملگرها) ۲.
std::vector<std::string> tokens;
std::string num;
for (char c : cleaned) {
    if (std::isdigit(static_cast<unsigned char>(c)) || c == '.') {
        num += c; // ساختن عدد (حتی اعشاری)
    } else {
        if (!num.empty()) { tokens.push_back(num); num.clear(); }
        // عددی را که رقم به رقم کانکت کرده ایم را در توکن ها پوش می کنیم
        tokens.push_back(std::string(1, c));
        // اضافه کردن عملگر - یعنی یک رشته با یک کاراکتر بساز که شامل کاراکتر سی باشد
    }
}

if (!num.empty()) tokens.push_back(num);
// اگر آخر رشته اکسپرشن یک عدد یک یا چند رقمه باشد، در حلقه بالا اضافه نمی شود و
// باید از اینجا اضافه کنیم
// الگوریتم Shunting-yard برای تبدیل به Postfix
std::stack<char> opStack;
std::ostringstream output;
// تعریف اولویت عملگرها
auto precedence = [](char op) {
    if (op == '*' || op == '/') return 2; // اولویت بالا
    if (op == '+' || op == '-') return 1; // اولویت پایین
    return 0;
};
bool first = true;

for (const auto& tok : tokens) {
    // اگر توکن یک عدد است، مستقیماً به خروجی می رود
    if (std::isdigit(static_cast<unsigned char>(tok[0])) || tok[0] == '.') {
        if (!first) output << " ";
        output << tok;
        first = false;
    }
    // اگر توکن پرانتز باز است، مستقیماً به پشته می رود
    else if (tok == "(") {
        opStack.push('(');
    }
    // اگر توکن پرانتز بسته است، عملگرها را تا رسیدن به پرانتز باز خارج می کنیم
    else if (tok == ")") {
        while (!opStack.empty() && opStack.top() != '(') {
            output << " " << opStack.top();
            opStack.pop();
        }
        if (!opStack.empty() && opStack.top() == '(') {
            opStack.pop(); // بدون افزودن به خروجی
        } else {
            throw std::runtime_error("Mismatched parentheses!");
        }
    }
    // اگر توکن یک عملگر است
    else if (tok == "+" || tok == "-" || tok == "*" || tok == "/") {
        // تا زمانی که پشته خالی نیست و عملگر بالای پشته '(' نیست و اولویتش <=
        // عملگر فعلی است، آن را خارج کن
        while (!opStack.empty() && opStack.top() != '(' &&
            precedence(opStack.top()) >= precedence(tok[0])) {
            output << " " << opStack.top();
            opStack.pop();
        }
        opStack.push(tok[0]); // عملگر فعلی را در پشته قرار بده
    }
}

```

```

}
// تخلیه عملگرهای باقی مانده در پشته، تا این نقطه باید پرانتزهای باز باید در
// مقابل پرانتزهای بسته خنثی شده باشند.
while (!opStack.empty()) {
    if (opStack.top() == '(' || opStack.top() == ')') {
        throw std::runtime_error("Mismatched parentheses!");
    }
    output << " " << opStack.top();
    opStack.pop();
}
return output.str();
}
// =====
// بخش ۳: تفسیر و محاسبه عبارت Postfix
// =====
std::unique_ptr<Expression> parsePostfix(const std::string& expr) {
    std::istringstream iss(expr);
    std::string token;
    std::stack<std::unique_ptr<Expression>> stack;
    while (iss >> token) {
        if (token == "+") {
            auto right = std::move(stack.top()); stack.pop();
            auto left = std::move(stack.top()); stack.pop();
            stack.push(std::make_unique<AddExpression>(std::move(left),
                                                        std::move(right)));
        }
        else if (token == "-") {
            auto right = std::move(stack.top()); stack.pop();
            auto left = std::move(stack.top()); stack.pop();
            stack.push(std::make_unique<SubtractExpression>(std::move(left),
                                                            std::move(right)));
        }
        else if (token == "*") {
            auto right = std::move(stack.top()); stack.pop();
            auto left = std::move(stack.top()); stack.pop();
            stack.push(std::make_unique<MultiplyExpression>(std::move(left),
                                                            std::move(right)));
        }
        else if (token == "/") {
            auto right = std::move(stack.top()); stack.pop();
            auto left = std::move(stack.top()); stack.pop();
            stack.push(std::make_unique<DivideExpression>(std::move(left),
                                                         std::move(right)));
        }
        else {
            // اگر عدد بود، آن را به درخت عبارت اضافه کن
            stack.push(std::make_unique<NumberExpression>(std::stod(token)));
        }
    }

    if (stack.size() != 1) {
        throw std::runtime_error("Invalid expression: Stack size is not 1 at the
                                end.");
    }
    return std::move(stack.top());
}
// =====
// بخش ۴: تابع اصلی (Main)
// =====
int main() {
    try {
        // مثال اصلی بدون پرانتز
        std::string infixExpr = "10 + 2 * 6 - 4 / 2";
        std::cout << "Main Expression (Infix): " << infixExpr << std::endl;
    }
}

```

```

// مرحله ۱: تبدیل به Postfix
std::string postfixExpr = infixToPostfix(infixExpr);
std::cout << "(Postfix): " << postfixExpr << std::endl;
// مرحله ۲: ساخت درخت عبارت و محاسبه
auto expr = parsePostfix(postfixExpr);
std::cout << "Tree Structure: " << expr->ToString() << std::endl;
std::cout << "Final Result: " << expr->Interpret() << std::endl;
// خروجی باید 20 باشد
std::cout << "-----" << std::endl;
// مثال جدید با پرانتز برای تغییر اولویت محاسبات
std::string infixExprWithParen = "(10 + 2) * 6 - 4 / 2";
std::cout << "Main Expression (Infix with parentheses): " <<
    infixExprWithParen << std::endl;
std::string postfixExprWithParen = infixToPostfix(infixExprWithParen);
std::cout << "(Postfix): " << postfixExprWithParen << std::endl;
auto exprWithParen = parsePostfix(postfixExprWithParen);
std::cout << "Tree Structure: " << exprWithParen->ToString() << std::endl;
std::cout << "Final Result: " << exprWithParen->Interpret() << std::endl;
// خروجی باید 70 باشد (12 * 6 - 2)
} catch (const std::exception& e) {
    std::cerr << "Error: " << e.what() << std::endl;
}
return 0;
}

```

مثالی کامل تر با قابلیت Parse کردن عبارات محاسباتی-منطقی و SQL با کمک الگوی Interpreter: 🎨🚀

```

#include <iostream>
#include <string>
#include <memory>
#include <map>
#include <stack>
#include <vector>
#include <sstream>
#include <variant>
#include <cctype>
#include <stdexcept>

// =====
// 0. Type Definitions & Utilities
// =====

using NumbersVariant = std::variant<int, float, double, long, long long,
    unsigned int, unsigned long, unsigned long long>;

void PrintVariant(const NumbersVariant& v) {
    std::visit([](auto&& arg) { std::cout << arg; }, v);
}

// =====
// 0.5 Unified Context
// =====

class Context {
    std::map<std::string, NumbersVariant> mathVars;
    std::map<std::string, bool> boolVars;
public:
    void SetMath(const std::string& key, NumbersVariant value) { mathVars[key] =
value; }
    void SetBool(const std::string& key, bool value) { boolVars[key] = value; }

    NumbersVariant GetMath(const std::string& key) const {
        auto it = mathVars.find(key);

```

```

        if (it != mathVars.end()) return it->second;
        throw std::runtime_error("Math variable not found: " + key);
    }

    bool GetBool(const std::string& key) const {
        auto it = boolVars.find(key);
        if (it != boolVars.end()) return it->second;
        throw std::runtime_error("Boolean variable not found: " + key);
    }
};

// =====
// 1. Math Expressions Interpreter
// =====

class Expression {
public:
    virtual ~Expression() = default;
    virtual NumbersVariant Interpret(const Context& ctx) const = 0;
    virtual std::string ToString() const = 0;
};

class NumberExpression : public Expression {
    NumbersVariant _value;
public:
    explicit NumberExpression(NumbersVariant value) : _value(value) {}

    NumbersVariant Interpret(const Context&) const override { return _value; }
    std::string ToString() const override {
        return std::visit([](auto&& arg) { return std::to_string(arg); }, _value);
    }
};

class VariableExpression : public Expression {
    std::string _name;
public:
    explicit VariableExpression(const std::string& name) : _name(name) {}
    NumbersVariant Interpret(const Context& ctx) const override {
        return ctx.GetMath(_name);
    }
    std::string ToString() const override { return _name; }
};

class MathBinaryOp : public Expression {
protected:
    std::unique_ptr<Expression> _left, _right;
    std::string _opSymbol;
public:
    MathBinaryOp(std::unique_ptr<Expression> l,
                 std::unique_ptr<Expression> r,
                 std::string op):
        _left(std::move(l)),
        _right(std::move(r)),
        _opSymbol(std::move(op)) {}
    std::string ToString() const override {
        return "(" + _left->ToString() + " " + _opSymbol + " " + _right->ToString() + " )";
    }
};

```



```

class AddExpression : public MathBinaryOp {
public:
    AddExpression(std::unique_ptr<Expression> l,
                  std::unique_ptr<Expression> r)
        : MathBinaryOp(std::move(l),
                        std::move(r),
                        "+") {}

    NumbersVariant Interpret(const Context& ctx) const override {
        auto l = _left->Interpret(ctx);
        auto r = _right->Interpret(ctx);
        return std::visit([](auto&& lhs, auto&& rhs) -> NumbersVariant { return lhs + rhs; }, l, r);
    }
};

class SubtractExpression : public MathBinaryOp {
public:
    SubtractExpression(std::unique_ptr<Expression> l,
                       std::unique_ptr<Expression> r)
        : MathBinaryOp(std::move(l),
                        std::move(r),
                        "-") {}

    NumbersVariant Interpret(const Context& ctx) const override {
        auto l = _left->Interpret(ctx);
        auto r = _right->Interpret(ctx);
        return std::visit([](auto&& lhs, auto&& rhs) -> NumbersVariant { return lhs - rhs; }, l, r);
    }
};

class MultiplyExpression : public MathBinaryOp {
public:
    MultiplyExpression(std::unique_ptr<Expression> l,
                       std::unique_ptr<Expression> r)
        : MathBinaryOp(std::move(l),
                        std::move(r),
                        "*") {}

    NumbersVariant Interpret(const Context& ctx) const override {
        auto l = _left->Interpret(ctx);
        auto r = _right->Interpret(ctx);
        return std::visit([](auto&& lhs, auto&& rhs) -> NumbersVariant { return lhs * rhs; }, l, r);
    }
};

class DivideExpression : public MathBinaryOp {
public:
    DivideExpression(std::unique_ptr<Expression> l,
                     std::unique_ptr<Expression> r)
        : MathBinaryOp(std::move(l),
                        std::move(r),
                        "/") {}

    NumbersVariant Interpret(const Context& ctx) const override {
        auto l = _left->Interpret(ctx);
        auto r = _right->Interpret(ctx);
        bool isZero = std::visit([](auto&& arg) -> bool { return arg == 0; }, r);
        if (isZero) throw std::runtime_error("Math Error: Division by zero!");
        return std::visit([](auto&& lhs, auto&& rhs) -> NumbersVariant { return lhs / rhs; }, l, r);
    }
};

class MathParser {
    struct Token {
        enum Type { NUMBER, VARIABLE, OP, LPAREN, RPAREN };

```

```

    Type type;
    std::string value;
};

static std::vector<Token> Tokenize(const std::string& expr) {
    std::vector<Token> tokens;
    for (size_t i = 0; i < expr.length(); ) {
        char c = expr[i];
        if (std::isspace(c)) { i++; continue; }
        if (std::isdigit(c) || c == '.') {
            std::string num;
            while (i < expr.length() && (std::isdigit(expr[i]) || expr[i] == '.')) num += expr[i++];
            tokens.push_back({Token::NUMBER, num});
        } else if (std::isalpha(c) || c == '_') {
            std::string var;
            while (i < expr.length() && (std::isalnum(expr[i]) || expr[i] == '_')) var += expr[i++];
            tokens.push_back({Token::VARIABLE, var});
        } else if (c == '+' || c == '-' || c == '*' || c == '/') {
            tokens.push_back({Token::OP, std::string(1, c)});
            i++;
        } else if (c == '(') { tokens.push_back({Token::LPAREN, "("}); i++; }
        else if (c == ')') { tokens.push_back({Token::RPAREN, ")"}); i++; }
        else { throw std::runtime_error("Invalid character in expression: " + std::string(1, c)); }
    }
    return tokens;
}

static int Precedence(const std::string& op) {
    if (op == "*" || op == "/") return 2;
    if (op == "+" || op == "-") return 1;
    return 0;
}

static std::unique_ptr<Expression> ApplyOp(const std::string& op,
                                           std::stack<std::unique_ptr<Expression>>& vals) {
    if (vals.size() < 2)
        throw std::runtime_error("Invalid expression: missing operands");
    auto right = std::move(vals.top()); vals.pop();
    auto left = std::move(vals.top()); vals.pop();
    if (op == "+")
        return std::make_unique<AddExpression>(std::move(left), std::move(right));
    if (op == "-")
        return std::make_unique<SubtractExpression>(std::move(left), std::move(right));
    if (op == "*")
        return std::make_unique<MultiplyExpression>(std::move(left), std::move(right));
    if (op == "/")
        return std::make_unique<DivideExpression>(std::move(left), std::move(right));
    throw std::runtime_error("Unknown operator: " + op);
}

public:
    static std::unique_ptr<Expression> Parse(const std::string& expression) {
        auto tokens = Tokenize(expression);
        std::stack<std::unique_ptr<Expression>> values;
        std::stack<std::string> ops;
        for (const auto& t : tokens) {
            if (t.type == Token::NUMBER) {
                if (t.value.find('.') != std::string::npos) //float
                    values.push(std::make_unique<NumberExpression>(std::stod(t.value)));
                else //int
                    values.push(std::make_unique<NumberExpression>(std::stoll(t.value)));
            } else if (t.type == Token::VARIABLE) {

```

```

        values.push(std::make_unique<VariableExpression>(t.value));
    } else if (t.type == Token::LPAREN) {
        ops.push("(");
    } else if (t.type == Token::RPAREN) {
        while (!ops.empty() && ops.top() != "(") {
            values.push(ApplyOp(ops.top(), values)); ops.pop();
        }
        if (!ops.empty()) ops.pop();
    } else if (t.type == Token::OP) {
        while (!ops.empty() && ops.top() != "(" &&
            Precedence(ops.top()) >= Precedence(t.value)) {
            values.push(ApplyOp(ops.top(), values)); ops.pop();
        }
        ops.push(t.value);
    }
}

while (!ops.empty()) { values.push(ApplyOp(ops.top(), values)); ops.pop(); }
if (values.size() != 1) throw std::runtime_error("Invalid expression structure");
return std::move(values.top());
}

};
// =====
// 2. Boolean Expressions Interpreter
// =====
class BooleanExpression {
public:
    virtual ~BooleanExpression() = default;
    virtual bool Interpret(const Context& ctx) const = 0;
    virtual std::string ToString() const = 0;
};

class BooleanVariable : public BooleanExpression {
    std::string _name;
public:
    explicit BooleanVariable(const std::string& name) : _name(name) {}
    bool Interpret(const Context& ctx) const override { return ctx.GetBool(_name); }
    std::string ToString() const override { return _name; }
};

class BooleanLiteral : public BooleanExpression {
    bool _value;
public:
    explicit BooleanLiteral(bool v) : _value(v) {}
    bool Interpret(const Context&) const override { return _value; }
    std::string ToString() const override { return _value ? "TRUE" : "FALSE"; }
};

class AndExpression : public BooleanExpression {
    std::unique_ptr<BooleanExpression> _left, _right;
public:
    AndExpression(std::unique_ptr<BooleanExpression> l,
        std::unique_ptr<BooleanExpression> r)
        : _left(std::move(l)),
          _right(std::move(r)) {}
    bool Interpret(const Context& ctx) const override {
        return _left->Interpret(ctx) && _right->Interpret(ctx);
    }
    std::string ToString() const override {
        return "(" + _left->ToString() + " AND " + _right->ToString() + ")";
    }
};

class OrExpression : public BooleanExpression {

```

```

    std::unique_ptr<BooleanExpression> _left, _right;
public:
    OrExpression(std::unique_ptr<BooleanExpression> l,
                 std::unique_ptr<BooleanExpression> r):
        _left(std::move(l)),
        _right(std::move(r)) {}
    bool Interpret(const Context& ctx) const override {
        return _left->Interpret(ctx) || _right->Interpret(ctx); }
    std::string ToString() const override {
        return "(" + _left->ToString() + " OR " + _right->ToString() + ")"; }
};

class NotExpression : public BooleanExpression {
    std::unique_ptr<BooleanExpression> _expr;
public:
    explicit NotExpression(std::unique_ptr<BooleanExpression> expr) :
        _expr(std::move(expr)) {}
    bool Interpret(const Context& ctx) const override {
        return !_expr->Interpret(ctx); }
    std::string ToString() const override { return "(NOT " + _expr->ToString() + ")"; }
};

class MathCompare : public BooleanExpression {
    std::unique_ptr<Expression> _left;
    std::unique_ptr<Expression> _right;
    std::string _op;
public:
    MathCompare(std::unique_ptr<Expression> l,
               std::unique_ptr<Expression> r,
               std::string op):
        _left(std::move(l)),
        _right(std::move(r)),
        _op(std::move(op)) {}

    bool Interpret(const Context& ctx) const override {
        auto l = _left->Interpret(ctx);
        auto r = _right->Interpret(ctx);
        return std::visit([&](auto&& a, auto&& b) -> bool {
            if (_op == ">") return a > b;
            if (_op == "<") return a < b;
            if (_op == ">=") return a >= b;
            if (_op == "<=") return a <= b;
            if (_op == "==") return a == b;
            if (_op == "!=") return a != b;
            return false;
        }, l, r);
    }
    std::string ToString() const override {
        return "(" + _left->ToString() + " " + _op + " " + _right->ToString() + ")"; }
};

class BoolParser {
    std::vector<std::string> _tokens;
    size_t _pos = 0;

    std::string Peek() const { return _pos < _tokens.size() ? _tokens[_pos] : ""; }
    std::string Consume() { return _tokens[_pos++]; }
    bool Match(const std::string& expected) {
        if (Peek() == expected) { _pos++; return true; }
        return false;
    }
}

```

```

static std::vector<std::string> TokenizeBool(const std::string& expr) {
    std::vector<std::string> tokens;
    std::string current;
    for (size_t i = 0; i < expr.length(); ++i) {
        char c = expr[i];
        if (std::isspace(c)) {
            if (!current.empty()) { tokens.push_back(current); current.clear(); }
        } else if (c == '(' || c == ')') {
            if (!current.empty()) { tokens.push_back(current); current.clear(); }
            tokens.push_back(std::string(1, c));
        } else if (c == '>' || c == '<' || c == '=' || c == '!') {
            if (!current.empty()) { tokens.push_back(current); current.clear(); }
            if (i + 1 < expr.length() && expr[i+1] == '=') {
                tokens.push_back(std::string(1, c) + "=");
                ++i;
            } else {
                tokens.push_back(std::string(1, c));
            }
        } else {
            current += c;
        }
    }
    if (!current.empty()) tokens.push_back(current);
    return tokens;
}

std::unique_ptr<BooleanExpression> ParseOr() {
    auto left = ParseAnd();
    while (Match("OR")) { auto right = ParseAnd();
        left = std::make_unique<OrExpression>(std::move(left), std::move(right)); }
    return left;
}

std::unique_ptr<BooleanExpression> ParseAnd() {
    auto left = ParseNot();
    while (Match("AND")) { auto right = ParseNot();
        left = std::make_unique<AndExpression>(std::move(left), std::move(right)); }
    return left;
}

std::unique_ptr<BooleanExpression> ParseNot() {
    if (Match("NOT")) return std::make_unique<NotExpression>(ParseNot());
    return ParseComparison();
}

std::unique_ptr<BooleanExpression> ParseComparison() {
    size_t startPos = _pos;
    std::string leftStr = "";
    int parenCount = 0; // شمارنده پرانتز برای تشخیص پرانتزهای ریاضی از بولین

    while (_pos < _tokens.size()) {
        std::string tok = Peek();
        if (tok == "(") {
            parenCount++;
        } else if (tok == ")") {
            if (parenCount == 0) break; // پایان عبارت ریاضی
            parenCount--;
        } else if (tok == ">" || tok == "<" || tok == ">=" || tok == "<="
            || tok == "==" || tok == "!=") {
            if (parenCount == 0) break; // عملگر مقایسه در سطح اصلی
        } else if (tok == "AND" || tok == "OR") {

```

```

        if (parenCount == 0) break; // عملگر بولین در سطح اصلی
    }
    leftStr += Consume() + " ";
}
std::string op = Peek();
if (op == ">" || op == "<" || op == ">=" || op == "<=" || op == "==" ||
    op == "!=") {
    Consume();
    std::string rightStr = "";
    parenCount = 0; // بازنشانی شمارنده برای سمت راست عملگر
    while (_pos < _tokens.size()) {
        std::string tok = Peek();
        if (tok == "(") {
            parenCount++;
        } else if (tok == ")") {
            if (parenCount == 0) break;
            parenCount--;
        } else if (tok == "AND" || tok == "OR") {
            if (parenCount == 0) break;
        }
        rightStr += Consume() + " ";
    }
    if (leftStr.empty() || rightStr.empty()) {
        throw std::runtime_error("Invalid comparison expression: missing operands");
    }

    auto leftExpr = MathParser::Parse(leftStr);
    auto rightExpr = MathParser::Parse(rightStr);
    return std::make_unique<MathCompare>(std::move(leftExpr),
        std::move(rightExpr), op);
}

// اگر عملگر مقایسه ای در سطح اصلی پیدا نشد، به عقب برگرد و به عنوان عبارت اولیه پردازش کن
_pos = startPos;
return ParsePrimary();
}

std::unique_ptr<BooleanExpression> ParsePrimary() {
    std::string tok = Peek();
    if (tok == "(") {
        Consume();
        auto expr = ParseOr();
        if (!Match("(")) throw std::runtime_error("Expected ')' in boolean expression");
        return expr;
    }
    if (tok == "TRUE") {
        Consume(); return std::make_unique<BooleanLiteral>(true); }
    if (tok == "FALSE") { Consume();
        return std::make_unique<BooleanLiteral>(false); }
    if (!tok.empty() && std::isalpha(tok[0])) {
        Consume();
        return std::make_unique<BooleanVariable>(tok);
    }
    throw std::runtime_error("Unexpected token in boolean expression: " + tok);
}

public:
    static std::unique_ptr<BooleanExpression> Parse(const std::string& expr) {
        BoolParser parser;
        parser._tokens = TokenizeBool(expr);
        auto result = parser.ParseOr();
        if (parser._pos < parser._tokens.size()) {

```

```

        throw std::runtime_error("Unexpected trailing tokens in boolean expression");
    }
    return result;
}
};
// =====
// 3. SQL Expressions Interpreter
// =====
class SQLExpression {
public:
    virtual ~SQLExpression() = default;
    virtual bool Evaluate(const std::map<std::string, std::string>& row) const = 0;
    virtual std::string ToString() const = 0;
};

class EqualsExpression : public SQLExpression {
    std::string _column, _value;

public:
    EqualsExpression(const std::string& c,
                     const std::string& v) :
        _column(c),
        _value(v) {}
    bool Evaluate(const std::map<std::string,
                                std::string>& row) const override {
        auto it = row.find(_column);
        return (it != row.end() && it->second == _value);
    }
    std::string ToString() const override { return _column + " = '" + _value + "'"; }
};

class GreaterThanExpression : public SQLExpression {
    std::string _column, _value;

public:
    GreaterThanExpression(const std::string& c,
                           const std::string& v) :
        _column(c),
        _value(v) {}
    bool Evaluate(const std::map<std::string,
                                std::string>& row) const override {
        auto it = row.find(_column);
        if (it != row.end()) {
            try { return std::stod(it->second) > std::stod(_value); }
            catch (...) { return it->second > _value; }
        }
        return false;
    }
    std::string ToString() const override { return _column + " > '" + _value + "'"; }
};

class SQLAndExpression : public SQLExpression {
    std::unique_ptr<SQLExpression> _left, _right;

public:
    SQLAndExpression(std::unique_ptr<SQLExpression> l,
                     std::unique_ptr<SQLExpression> r) :
        _left(std::move(l)),
        _right(std::move(r)) {}
    bool Evaluate(const std::map<std::string, std::string>& row) const override {
        return _left->Evaluate(row) && _right->Evaluate(row);
    }
};

```

```

    std::string ToString() const override {
        return "(" + _left->ToString() + " AND " + _right->ToString() + ")"; }
};

class SimpleDatabase {
    std::vector<std::map<std::string, std::string>> _data;
public:
    void AddRow(const std::map<std::string, std::string>& row) {
        _data.push_back(row); }

    std::vector<std::map<std::string, std::string>>
        Query(std::unique_ptr<SQLExpression> whereClause) const {
        std::vector<std::map<std::string, std::string>> result;
        for (const auto& row : _data) {
            if (whereClause->Evaluate(row)) result.push_back(row);
        }
        return result;
    }
};

// =====
// 4. Postfix Parser
// =====

std::unique_ptr<Expression> ParsePostfix(const std::string& expr) {
    std::istringstream iss(expr);
    std::string token;
    std::stack<std::unique_ptr<Expression>> stack;
    while (iss >> token) {
        if (token == "+") {
            if (stack.size() < 2)
                throw std::runtime_error("Postfix error: not enough operands for +");
            auto r = std::move(stack.top()); stack.pop();
            auto l = std::move(stack.top()); stack.pop();
            stack.push(std::make_unique<AddExpression>(std::move(l), std::move(r)));
        } else if (token == "-") {
            if (stack.size() < 2)
                throw std::runtime_error("Postfix error: not enough operands for -");
            auto r = std::move(stack.top()); stack.pop();
            auto l = std::move(stack.top()); stack.pop();
            stack.push(std::make_unique<SubtractExpression>(std::move(l),
                std::move(r)));
        } else if (token == "*") {
            if (stack.size() < 2)
                throw std::runtime_error("Postfix error: not enough operands for *");
            auto r = std::move(stack.top()); stack.pop();
            auto l = std::move(stack.top()); stack.pop();
            stack.push(std::make_unique<MultiplyExpression>(std::move(l),
                std::move(r)));
        } else if (token == "/") {
            if (stack.size() < 2)
                throw std::runtime_error("Postfix error: not enough operands for /");
            auto r = std::move(stack.top()); stack.pop();
            auto l = std::move(stack.top()); stack.pop();
            stack.push(std::make_unique<DivideExpression>(std::move(l),
                std::move(r)));
        } else {
            if (token.find('.') != std::string::npos)
                stack.push(std::make_unique<NumberExpression>(std::stod(token)));
            else stack.push(std::make_unique<NumberExpression>(std::stoll(token)));
        }
    }
}

```



```

    if (stack.size() != 1)
        throw std::runtime_error("Invalid postfix expression structure");
    return std::move(stack.top());
}
// =====
// 5. Usage Examples
// =====
void Example1() {
    std::cout << "=== Example 1: Unified Interpreter System (Math + Boolean + Comparison) ==="
                << std::endl;
    Context ctx;
    ctx.SetMath("x", 10); ctx.SetMath("y", 5.5); ctx.SetMath("z", 2);
    ctx.SetMath("zero_var", 0); ctx.SetMath("score", 85);
    ctx.SetBool("is_active", true); ctx.SetBool("has_permission", false);
    ctx.SetBool("is_admin", true);

    std::cout << "=====\n Unified Interpreter
    System Demo\n=====\n\n";
    std::string mathExprStr = "x * z + y / 2";
    std::cout << "[Math Engine]\nExpression : " << mathExprStr << "\n";
    auto mathAst = MathParser::Parse(mathExprStr);
    std::cout << "AST Tree : " << mathAst->ToString() << "\nResult : ";
    PrintVariant(mathAst->Interpret(ctx));
    std::cout << "\n\n";

    std::string boolExprStr = "is_active AND NOT has_permission";
    std::cout << "[Boolean Engine - Pure]\nExpression : " << boolExprStr << "\n";
    auto boolAst = BoolParser::Parse(boolExprStr);
    std::cout << "AST Tree : " << boolAst->ToString() << "\nResult : " <<
        std::boolalpha << boolAst->Interpret(ctx) << "\n\n";

    std::string combinedExprStr = "score >= 80 AND is_active AND NOT is_admin";
    std::cout << "[Combined Engine - Math Comparison + Boolean]\nExpression : "
                << combinedExprStr << "\n";
    auto combinedAst = BoolParser::Parse(combinedExprStr);
    std::cout << "AST Tree : " << combinedAst->ToString() << "\nResult : "
                << std::boolalpha << combinedAst->Interpret(ctx) << "\n\n";
    std::cout << "[Comparison Operators Stress Test]\n";
    std::vector<std::string> tests = {
        "x > 5",
        "y <= 5.5",
        "z == 2",
        "x != 0",
        "(x * z) > 15 AND y < 10"
    };
    for (const auto& test : tests) {
        auto ast = BoolParser::Parse(test);
        std::cout << " Test : " << test << "\n AST : " << ast->ToString()
                    << "\n Result : " << std::boolalpha << ast->Interpret(ctx) <<
                    "\n ----- \n";
    }
}

void Example2() {
    std::cout << "\n=== Example 2: Math Expressions Interpreter ===" << std::endl;
    Context ctx;
    ctx.SetMath("x", 5); ctx.SetMath("y", 10); ctx.SetMath("z", 2);
    try {
        std::string exprStr = "x + y * z";
        std::cout << "Parsing expression: '" << exprStr << "'" << std::endl;
        auto expression = MathParser::Parse(exprStr);
    }
}

```

```

        std::cout << "AST Tree      : " << expression->ToString() << std::endl;
        std::cout << "Result        : ";
        PrintVariant(expression->Interpret(ctx));
        std::cout << "\n" << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Error in Example2: " << e.what() << "\n" << std::endl;
    }
}

void Example3() {
    std::cout << "=== Example 3: Boolean Expressions Interpreter ===" << std::endl;
    Context ctx;
    ctx.SetBool("A", true); ctx.SetBool("B", false); ctx.SetBool("C", true);
    try {
        auto expr1 = std::make_unique<AndExpression>( std::make_unique<BooleanVariable>("A"),
                                                    std::make_unique<BooleanVariable>("B"));
        auto expr2 = std::make_unique<NotExpression>(std::make_unique<BooleanVariable>("C"));
        auto finalExpr = std::make_unique<OrExpression>(std::move(expr1),
                                                         std::move(expr2));

        std::cout << "Expression  : " << finalExpr->ToString() << std::endl;
        std::cout << "Result        : " << std::boolalpha << finalExpr->Interpret(ctx)
                    << "\n" << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Error in Example3: " << e.what() << "\n" << std::endl;
    }
}

void Example4() {
    std::cout << "=== Example 4: SQL Expressions Interpreter ===" << std::endl;
    try {
        SimpleDatabase db;
        db.AddRow({{"name", "Alice"}, {"age", "25"}, {"city", "New York"}});
        db.AddRow({{"name", "Bob"}, {"age", "30"}, {"city", "London"}});
        db.AddRow({{"name", "Charlie"}, {"age", "22"}, {"city", "Paris"}});
        db.AddRow({{"name", "David"}, {"age", "35"}, {"city", "Tokyo"}});

        auto condition = std::make_unique<SQLAndExpression>(
            std::make_unique<GreaterThanExpression>("age", "25"),
            std::make_unique<EqualsExpression>("city", "London")
        );

        std::cout << "Search condition: " << condition->ToString() << std::endl;
        auto results = db.Query(std::move(condition));

        std::cout << "Results:" << std::endl;
        if (results.empty()) std::cout << " (No records found)" << std::endl;
        for (const auto& row : results) {
            std::cout << "  Name: " << row.at("name") << ", Age: " << row.at("age")
                        << ", City: " << row.at("city") << std::endl;
        }
        std::cout << std::endl;
    } catch (const std::exception& e) {
        std::cerr << "Error in Example4: " << e.what() << "\n" << std::endl;
    }
}

// =====
// 6. Main Function
// =====

int main(int argc, char *argv[]) {
    try {
        std::cout << "--- Test ParsePostfix Run: ---" << std::endl;
        auto expr = ParsePostfix("5 3 + 2 -");
    }
}

```

```

Context emptyContext;
std::cout << "Result: ";
PrintVariant(expr->Interpret(emptyContext));
std::cout << "\n" << std::endl;

std::cout << "--- Interpreter Examples ---" << std::endl;
Example1();
Example2();
Example3();
Example4();
} catch (const std::exception& e) {
    std::cerr << "Fatal Error: " << e.what() << std::endl;
}
std::cout<<"=====END===== "<<std::endl;
return 0;
}

```

نتیجه: 

--- Test ParsePostfix Run: ---

Result: 6

--- Interpreter Examples ---

=== Example 1: Unified Interpreter System (Math + Boolean + Comparison) ===

=====

Unified Interpreter System Demo

=====

[Math Engine]

Expression : $x * z + y / 2$

AST Tree : $((x * z) + (y / 2))$

Result : 22.75

[Boolean Engine - Pure]

Expression : `is_active AND NOT has_permission`

AST Tree : `(is_active AND (NOT has_permission))`

Result : true

[Combined Engine - Math Comparison + Boolean]

Expression : `score >= 80 AND is_active AND NOT is_admin`

AST Tree : `(( (score >= 80 ) AND is_active) AND (NOT is_admin))`

Result : false

[Comparison Operators Stress Test]

Test : $x > 5$

AST : $(x > 5)$

Result : true

Test : $y \leq 5.5$

AST : $(y \leq 5.500000)$

Result : true

```

-----
Test   : z == 2
AST    : ( z == 2 )
Result : true
-----

Test   : x != 0
AST    : ( x != 0 )
Result : true
-----

Test   : ( x * z ) > 15 AND y < 10
AST    : (( ( x * z ) > 15 ) AND ( y < 10 ))
Result : true
-----

```

=== Example 2: Math Expressions Interpreter ===

```

Parsing expression: 'x + y * z'
AST Tree   : ( x + ( y * z ) )
Result     : 25

```

=== Example 3: Boolean Expressions Interpreter ===

```

Expression : ((A AND B) OR (NOT C))
Result     : false

```

=== Example 4: SQL Expressions Interpreter ===

```

Search condition: (age > '25' AND city = 'London')
Results:
  Name: Bob, Age: 30, City: London

```

=====END=====

چالش اصلی تبدیل عبارت محاسباتی به الگوی Interpreter تبدیل محاسبات نوع Infix که قابل فهم برای انسانها است (مثل $5 + 3 - 8$) به نوع Postfix است که قابل فهم برای کامپیوتر است (مثل $5\ 3\ +\ 8$)! چرا مدل Postfix برای کامپیوترها ساده است؟ چون به سادگی با خواندن توکن هایی که با یک space از هم جدا شده اند (اعداد و علامت ها از چپ به راست، هر بند یا Token (که شامل اعداد یا علامت ها هستند) را می خوانید و به هر عدد که رسیدید، آنرا در پشته اعداد push می کنید و به هر علامتی که رسیدید، دو عدد بالای پشته اعداد را pop می کنید و بر اساس علامت کنونی، محاسبه بین این دو عدد را انجام داده و نتیجه را به همان پشته اعداد push می کنید و پیش می روید و در این مرحله دیگر درگیر پرانتز و اولویت های دیگر نیستید چرا که در مرحله قبل، علامت ها را بر اساس اولویت علامت ها و همچنین بر اساس پرانتزها به درستی تنظیم کرده اید! به جهت اینکه مساله تبدیل عبارت های Infix به Postfix مساله الگوریتمیک است و فراتر از موضوع این کتاب است، از توضیح اضافی در این مورد پرهیز می کنیم! مثال اول Parse کردن Expression های محاسبات ریاضی ساده تر است و با درک آن، می توانید رویکرد کامل تر محاسباتی را هم درک کنید! مثال دوم مثالی با انواع داده های عددی و محاسباتی و منطقی و SQL است و فراتر از محاسبات ریاضی است!

Pattern	کاربرد
Composite	ساخت درخت Expression
Visitor	عملیات‌های جدید روی Expression
Builder	ساخت Parser
Flyweight	بهینه‌سازی حافظه
Strategy	انتخاب نوع

جمع‌بندی

✓ الگوی Interpreter برای موقعیت‌هایی مناسب است که نیاز به پردازش دستورات یا عبارات محاسباتی یا ... دارید. با وجود سادگی مفهوم، برای دستور زبان‌های پیچیده ممکن است باعث ایجاد تعداد زیادی کلاس شود. در چنین مواردی بهتر است از ابزارهایی مانند parser generator ها استفاده کنید.

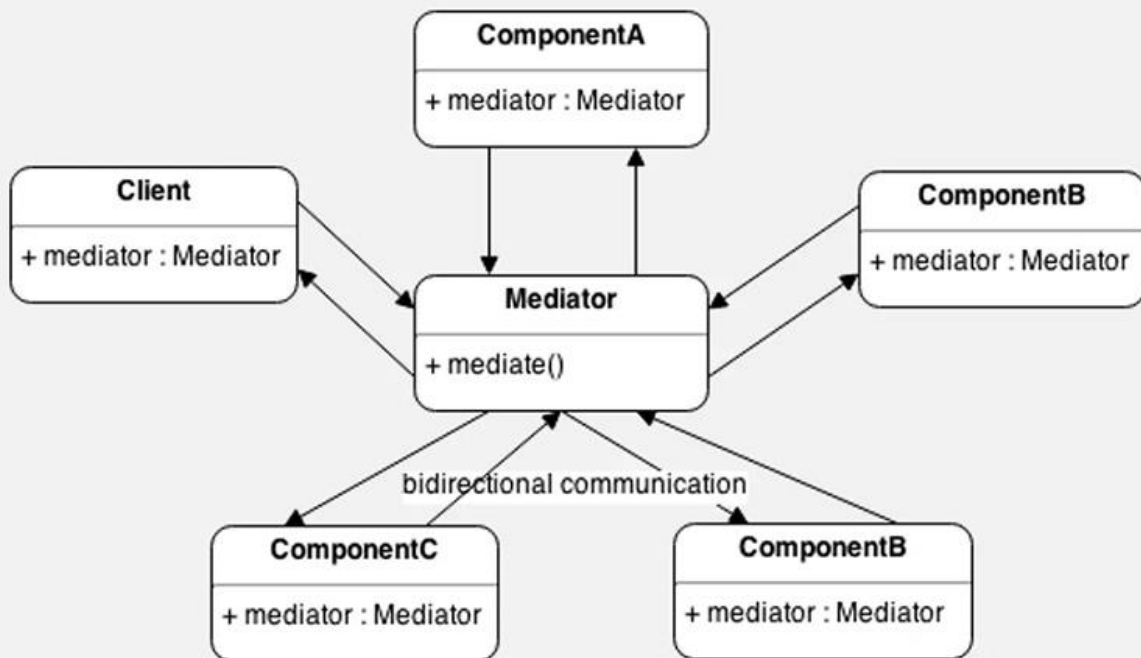
✓ Interpreter برای زبان‌های کوچک و قابل توسعه عالی است!

✓ برای Rule Engine ها فوق‌العاده کاربردی است!

✓ در C++ با unique_ptr و Polymorphism بسیار تمیز پیاده‌سازی می‌شود!



Mediator pattern – Class diagram



این پترن، ارتباط دهنده گروه های مختلفی از اشیای مرتبط با هم به صورت غیر مستقیم با حذف **Coupling** و درگیری بین آنها است. به نوعی شاید بتوان آنرا شبیه به کنترلر در مدل **MVC** دانست! یا می توان باند فرود فرودگاهها یا چراغ راهنمایی در چهارراهها را مثال عملی آن دانست که با کنترل ترافیک هواپیماها یا خودروها و زمان بندی بین حرکت وسایل نقلیه تلاش دارند مسیرهای زمینی را با زمان بندی در اختیار تمام وسایل نقلیه ای که به این مسیر نیاز دارند قرار بدهند به گونه ای که تصادف وسایل نقلیه پدید نیاید! در این پترن قرار نیست اشیای به هم مرتبط، از موجودیت هم خبردار شوند، فقط قرار است هر کدام کاری را انجام دهند و **Notify** کنند و این پیامها از میان یک **Mediator** رد شوند و با مدیریت **Mediator**، دیگرانی که منتظر خبر مربوطه هستند، با شنیدن خبر، کار مناسب مرتبط با آنرا انجام دهند!

✅ مزایای استفاده از Meiator:

- حذف Coupling
- مدیریت واحد ارتباطات بین بخش های مختلف
- سادگی در مدیریت و توسعه

❌ معایب استفاده از Mediator:

- پیچیده تر شدن خود کلاس **Mediator**
- هماهنگی و وابستگی همه بخش ها به **Mediator** واسط باعث می شود خود **Mediator** به **God Object** که همه به آن وابسته هستند تبدیل شود! خود **God Object** نوعی **Anti Pattern** محسوب می شود!

🎨 مثال: می خواهیم در ++C با استفاده از **Mediator** یک چت روم بسازیم تا کاربران به هم پیام دهند!

```
#include <iostream>
#include <string>
#include <vector>
#include <memory>
// Forward declaration
class User;
// Mediator Interface
class ChatMediator {
public:
    virtual void sendMessage(const std::string& message, User* sender) = 0;
    virtual void sendMessageTo(const std::string& message, User* sender, User*
receiver) = 0;
    virtual void addUser(User* user) = 0;
    virtual ~ChatMediator() = default;
};
// Colleague
class User {
protected:
    ChatMediator* mediator;
    std::string name;

public:
    User(const std::string& name, ChatMediator* med) : name(name), mediator(med) {}
    virtual void send(const std::string& message) {
        mediator->sendMessage(message, this);
    }
}
```

```

    virtual void sendTo(const std::string& message, User* Receiver) {
        mediator->sendMessageTo(message, this, Receiver);
    }

    virtual void receive(const std::string& message, const std::string& senderName) {
        std::cout << name << " received from " << senderName << ": " << message << std::endl;
    }
    std::string getName() const { return name; }
};
// Concrete Mediator
class ChatRoom : public ChatMediator {
    std::vector<User*> users;

public:
    void addUser(User* user) override {
        users.push_back(user);
    }

    void sendMessage(const std::string& message, User* sender) override {
        for (auto* user : users) {
            if (user != sender) {
                user->receive(message, sender->getName());
            }
        }
    }
    void sendMessageTo(const std::string& message, User* sender, User* to) override
    {
        for (auto* user : users) {
            if (user == to) {
                user->receive(message, sender->getName());
            }
        }
    }
};

```

پیاده سازی: 

```

int main() {
    ChatRoom room;

    User user1("Alice", &room);
    User user2("Bob", &room);
    User user3("Charlie", &room);

    room.addUser(&user1); room.addUser(&user2); room.addUser(&user3);

    user1.send("Hello everyone!");
    user2.send("Hi Alice!");

    return 0;
}

```


❁ پیاده سازی الگوی Mediator به عنوان Command Bus در لایه 3 از معماری Clean در مایکروسرویس ها:

توضیحات تکمیلی الگوی Clean را در فصل مختص خودش داده ایم، ولی به جهت آشنایی با روش پیش رو، ما کلاسی به نام Command که از نوع ICommand است بوجود می آوریم و برای انواع Command هایی که ممکن است یک کلاس به کلاس دیگری ارسال کند، یک کلاس Command خاص برای آن می سازیم و در نتیجه، کلاسهای مختلف به جای پیاده سازی انواع توابع با ورودی های مختلف، به سادگی به واسطه ICommand به هم Command با داشتن Property های مختلف ارسال می کنند! دریافت کننده ها نیز Command ورودی را به صورت مشتق شده دریافت می کنند و داده های مختص نوع Command اختصاصی خود را از Command ارسال شده توسط ارسال کننده می گیرند!

```
//1. Command Base Class
struct ICommand {
    virtual ~ICommand() = default;
};

//2. Concrete Command - دستور ساخت سفارش - شی مشتق شده از آی کامند
struct CreateOrderCommand : public ICommand {
    std::string product;
    int quantity;
};

//3. Handler Interface - اینترفیس سازنده سفارش با دریافت نوعی از کامند ساخت سفارش
template<typename TCommand>
class ICommandHandler {
public:
    virtual void handle(const TCommand& command) = 0;
    virtual ~ICommandHandler() = default;
};

//4. Concrete Handler
class CreateOrderHandler : public ICommandHandler<CreateOrderCommand> {
public:
    void handle(const CreateOrderCommand& command) override {
        std::cout<<"Creating order in db for "<<command.quantity<<" x "<<command.product<<"\n";
    }
};

//5. CommandBus (Mediator)
#include <unordered_map>
#include <functional>
#include <typeindex>
#include <memory>

class CommandBus {
    std::unordered_map<std::type_index, std::function<void(const ICommand&)>> handlers;
public:
    template<typename TCommand>
    void registerHandler(std::shared_ptr<ICommandHandler<TCommand>> handler) {
        handlers[typeid(TCommand)] = [handler](const ICommand& command) {
            handler->handle(static_cast<const TCommand&>(command));
        };
    }

    void send(const ICommand& command) {
        auto it = handlers.find(typeid(command));
        if (it != handlers.end()) {
            it->second(command);
        } else {
            std::cerr << "No handler found for command.\n";
        }
    }
};
```

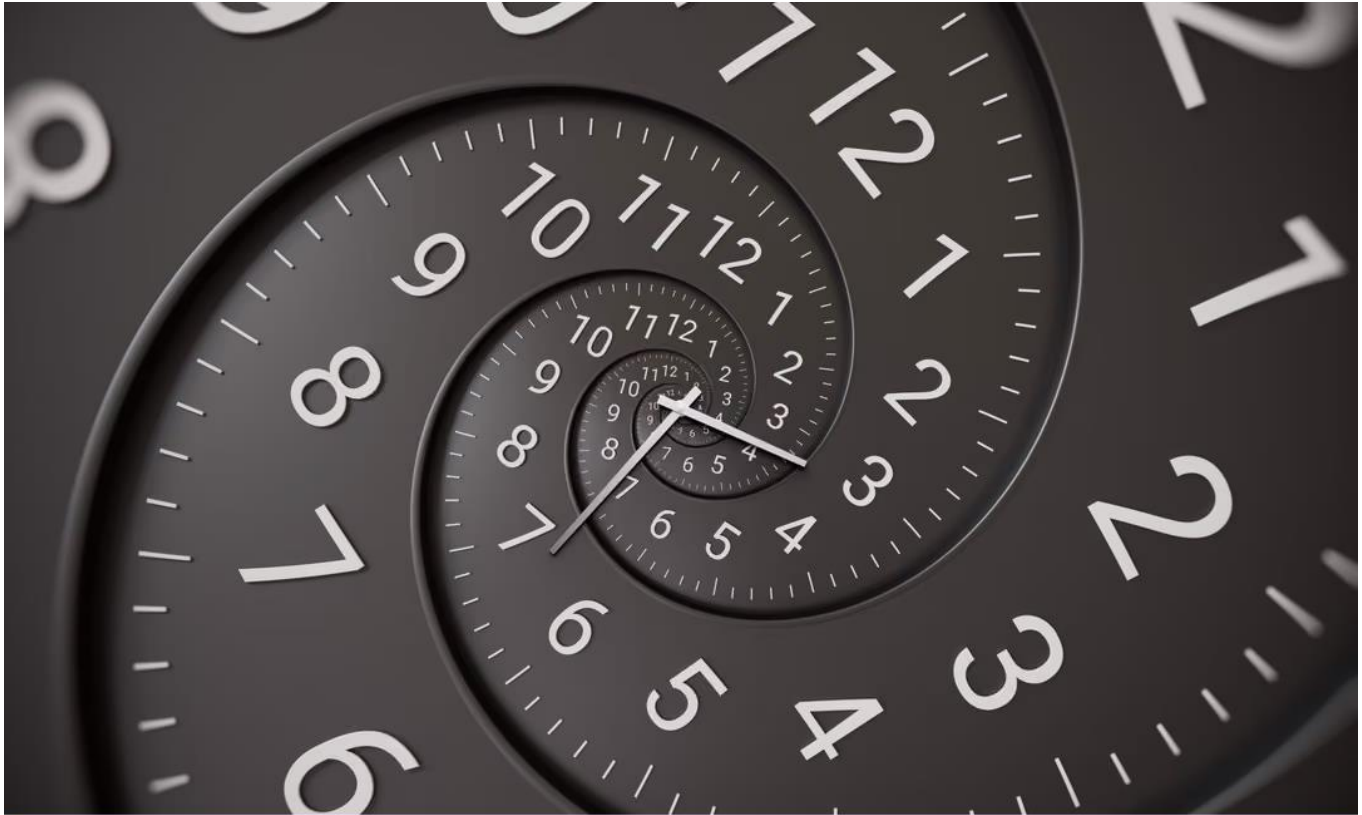
```
//6. Usage
int main() {
    CommandBus bus;

    auto createHandler = std::make_shared<CreateOrderHandler>();
    bus.registerHandler<CreateOrderCommand>(createHandler);

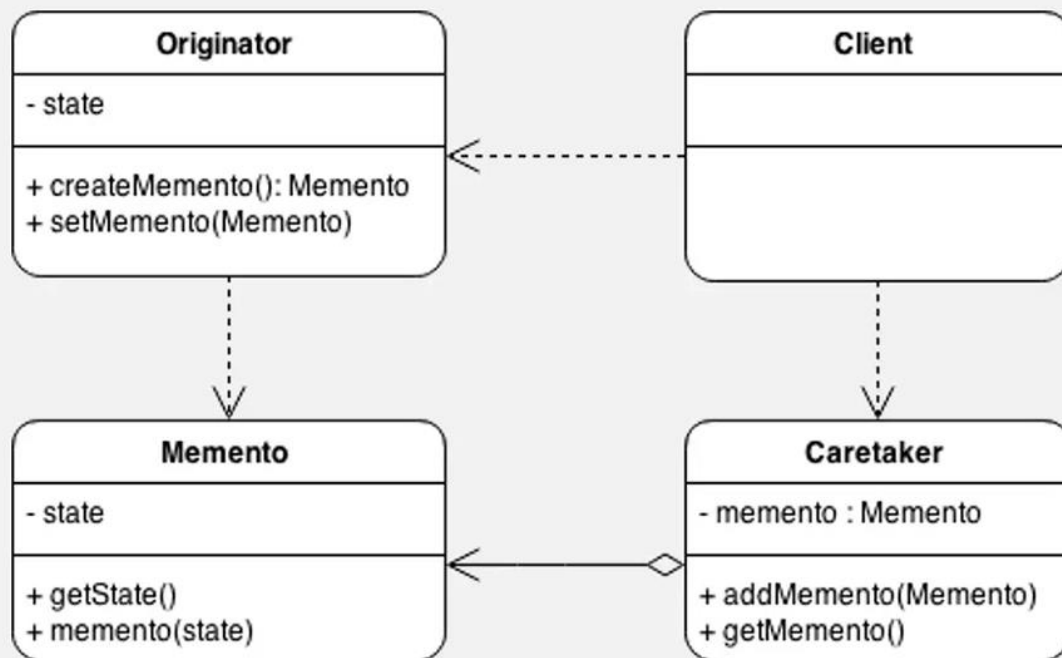
    CreateOrderCommand cmd{ "Book", 3 };
    bus.send(cmd); // Will invoke CreateOrderHandler
}
```

📌 در نتیجه در یک میکروسرویس با معماری Clean، Command Bus، Mediator است، مدیر مرکزی ارتباطات بخش های مختلف (Aggregate Root های مختلف) است! CommandBusHandler ها هم با لایه اینترفیس، تزریق شونده هستند! اینجا می توان معماری Clean را به خوبی پیاده کرد چون کنترلر چیزی از منطق استفاده نمی داند و فقط یک Command را به bus ارسال می کند و بخش مربوطه کار مورد نظر را انجام می دهد! در مثال فوق ما کلاسی را تعریف کردیم که با ورودی Command مورد نیازش، سرویسی مانند ساخت یک Order در دیتابیس را انجام می دهد! پس ما کلاس مد نظر را ساختیم و در Command Bus نوع Mediator آنرا همراه با Command مورد نیازش به عنوان Template توسط متد Register ثبت کردیم و از این به بعد، با ساخت یک Command از نوع مد نظر (مانند Command از نوع CreateOrderCommand) و Send کردن آن به Command Bus به نوعی همان دستور ساخت یک Order جدید را بدون آگاهی از روند ساخت Order اجرا کردیم! در اینجا Command Bus ما به عنوان Mediator بر اساس نوع Command به سراغ Handler مختص ثبت شده آن یعنی CreateCommandHandler رفت و آنرا از Hash Table مربوطه پیدا کرد و Command ارسالی را به Handler یافته شده داد و در نتیجه Handler هم دستور ارسالی را اجرا کرد! شما می توانید با این ساختار، به خوبی معماری CQRS را هم اجرایی کنید و Read Command ها (Query) و Write Command ها (Command) را از طریق Command Bus ارسال کنید! به بیان بهتر برای Read Command ها، داده های نوع Query را به Bus ارسال کنید تا handler-های مختص Read Command ها را یافته و عمل واکنشی اطلاعات را انجام دهند و با ارسال Command ها نیز Write Command Handler ها را بیابد و با واسطه Write Command Handler یافته شده، درخواست ویرایش اطلاعات شما را اجرایی کند! البته در بخش مربوط به معماری CQRS به صورت کامل به توضیح این معماری پرداخته ایم و از اشاره بیشتر در این بخش پرهیز می کنیم!

○ **Memento**: پترنی از نوع Behavioral است! بازگرداندن وضعیت قبلی اشیا، بدون نقض اصل Encapsulation



Memento pattern – Class diagram



در فرآیند طراحی نرم‌افزار، گاهی لازم است که وضعیت قبلی یک شیء را ذخیره کنیم تا بتوانیم در آینده آن را بازگردانیم. مثلاً در یک ویرایشگر متن (Editor)، کاربر ممکن است چند بار تغییر انجام دهد و سپس بخواهد با زدن دکمه‌ی Undo، به حالت قبلی برگردد. در سطحی بالاتر شاید بتوان سورس کنترلرها را نیز جزو این کتگوری دانست چرا که سورس شما به تدریج تکمیل می‌شود و می‌توانید سورس را به نسخه‌های عقب‌تر هم ببرید و یا نسخه فعلی Repositoryتان را از نسخه‌های قبلی بازیابی نمایید!

اما طبق اصل Encapsulation در شیء‌گرایی، نباید از بیرون به وضعیت درونی یک شیء دسترسی مستقیم داشته باشیم. پس چگونه می‌توان وضعیت را ذخیره و بازگرداند؟ در اینجا، الگوی طراحی Memento Pattern وارد عمل می‌شود.

ایده‌ی اصلی

الگوی Memento به ما اجازه می‌دهد state داخلی یک شیء را بدون شکستن اصل encapsulation ذخیره کنیم، تا بعداً بتوان آن را بازیابی کرد. این الگو سه نقش اصلی دارد:

Originator ← شیئی که وضعیتش باید ذخیره یا بازیابی شود!

Memento ← شیئی که وضعیت داخلی Originator را نگه می‌دارد.

Caretaker ← مسئول نگهداری Memento ها (تاریخچه‌ی وضعیت‌ها).

 مثال: شبیه‌سازی یک Text Editor با قابلیت Undo: در این مثال، یک Editor داریم که شامل دو مقدار است:

title

content

هر بار که کاربر تغییری در Editor ایجاد می‌کند، باید وضعیت فعلی ذخیره شود تا بتواند در آینده به آن برگردد. (Undo)

کلاس Memento: EditorState

```
#pragma once
#include <string>

class EditorState {
    std::string title;
    std::string content;

public:
    EditorState(std::string t, std::string c)
        : title(std::move(t)), content(std::move(c)) {}

    std::string getTitle() const { return title; }
    std::string getContent() const { return content; }
};
```

کلاس EditorState همان Memento است. این کلاس وضعیت داخلی Editor را ذخیره می‌کند، اما اجازه‌ی تغییر آن را به بیرون نمی‌دهد.

```
#pragma once
#include <string>
#include <memory>
#include "EditorState.h"

class Editor {
    std::string title;
    std::string content;
public:
    void setTitle(const std::string& t) { title = t; }
    void setContent(const std::string& c) { content = c; }
    std::string getTitle() const { return title; }
    std::string getContent() const { return content; }

    std::shared_ptr<EditorState> createState() const {
        return std::make_shared<EditorState>(title, content);
    }

    void restore(const std::shared_ptr<EditorState>& state) {
        title = state->getTitle();
        content = state->getContent();
    }
};
```

کلاس Editor همان **Originator** است. او می‌تواند با متد `createState()` وضعیت فعلی‌اش را در قالب یک **Memento** بسازد و با متد `restore()`، از یک **Memento** برای بازگرداندن حالت قبلی استفاده کند.

```
#pragma once
#include <vector>
#include <memory>
#include "EditorState.h"
class History {
    std::vector<std::shared_ptr<EditorState>> states;
public:
    void push(std::shared_ptr<EditorState> state) {
        states.push_back(std::move(state));
    }

    std::shared_ptr<EditorState> pop() {
        if (states.empty()) return nullptr;
        auto last = states.back();
        states.pop_back();
        return last;
    }

    bool empty() const { return states.empty(); }
};
```

کلاس History همان **Caretaker** است. این کلاس مسئول نگهداری لیست از وضعیت‌های ذخیره‌شده **Memento** ها است و با `push()` و `pop()` مدیریت undo را ممکن می‌سازد!

```
#include <iostream>
#include "Editor.h"
#include "History.h"
int main() {
    Editor editor;
    History history;

    editor.setTitle("First Draft");
    editor.setContent("This is the first version.");
    history.push(editor.createState()); // Save state

    editor.setContent("This is the second version, with some edits.");
    history.push(editor.createState()); // Save state

    editor.setContent("Third version, with even more changes.");
    std::cout << "Current content: " << editor.getContent() << "\n";
    // Undo 1
    auto prev = history.pop();
    if (prev) editor.restore(prev);
    std::cout << "After undo: " << editor.getContent() << "\n";
    // Undo 2
    prev = history.pop();
    if (prev) editor.restore(prev);
    std::cout << "After second undo: " << editor.getContent() << "\n";

    return 0;
}
```

خروجی: 

Current content: Third version, with even more changes.

After undo: This is the second version, with some edits.

After second undo: This is the first version.

در این ساختار، هیچ کلاسی به صورت مستقیم به جزئیات وضعیت داخلی Editor دسترسی ندارد، در نتیجه اصل **Encapsulation** رعایت شده است.

مزایا: 

-  **Encapsulation** حفظ می‌شود: وضعیت داخلی Editor فقط درون خودش تعریف می‌شود.
-  **Undo/Redo** ساده: با **push** و **pop** روی **history** می‌توان حالت‌ها را به سادگی مدیریت کرد.
-  قابل گسترش: می‌توان متدهای **Redo** یا محدودیت روی تعداد **state** ها اضافه کرد.

🔑 در چه مواردی از Memento Pattern استفاده می‌شود؟

1. وقتی نیاز دارید وضعیت قبلی یک شیء را ذخیره کنید تا بتوانید بعداً آن را بازیابی کنید. (Undo/Redo).
2. وقتی می‌خواهید این کار را بدون افشای جزئیات درونی شیء انجام دهید.

🎨 مثال‌های دیگر

- ویرایشگر گرافیکی برای Undo آخرین تغییر در تصویر.
- بازی‌های ویدیویی برای ذخیره‌ی حالت بازی. (Save/Load)
- فرم‌های وب یا اپلیکیشن‌ها برای بازگردانی تنظیمات کاربر.
- سیستم‌های مالی که لازم است snapshot از وضعیت حساب کاربر ذخیره شود.

🔄 وجه اشتراک و تفاوت الگوهای Command و Memento:

وجه اشتراک الگوهای Command و Memento در مدیریت تغییرات وضعیت (State) اشیاء در طول زمان و پیاده‌سازی قابلیت‌هایی مانند Undo/Redo است. اما تفاوت بنیادین آن‌ها در چیست؟

الگوی Command برای عملیات‌های قابل بازگشت از طریق محاسبه معکوس مناسب است. مثلاً برای جابه‌جایی یک شیء، به جای ذخیره مختصات دقیق، یک Command به نام Move می‌سازیم که فقط مقدار تغییرات (مثلاً dx و dy) را در خود دارد. با اجرای معکوس این دستور (Undo) به حالت قبل و با اجرای مجدد (Redo) به حالت بعد می‌رسیم. این رویکرد برای محاسبات ریاضی یا تغییرات مختصات که فرمول بازگشت دارند، بسیار کارآمد است؛ زیرا به جای ذخیره وضعیت‌های مختلف، فقط «دستور» و «نحوه خنثی کردن آن» را نگه می‌داریم.

اما اگر عملیات بازگشت‌ناپذیر یا بسیار پیچیده باشد چه؟ فرض کنید کاربر در حال تایپ کردن است و کاراکترها مدام جایگزین می‌شوند، یا اینکه یک فیلتر سنگین روی تصویر اعمال شده است. یا فرض کنید از داده خاصی Hash غیر قابل بازگشت به مقدار اصلی می‌گیرد و در این شرایط، محاسبه معکوس عملیات عملاً غیرممکن یا بسیار پرهزینه است. اینجا است که الگوی Memento وارد عمل می‌شود. Memento به جای ذخیره "دستور"، یک "عکس فوری" (Snapshot) از وضعیت کامل شیء را در لحظه قبل ذخیره می‌کند تا در صورت نیاز، دقیقاً و بدون هیچ محاسبه‌ای به همان حالت بازگردیم.

در نهایت، در نرم‌افزارهای پیچیده‌ای مانند فتوشاپ، می‌توان این دو الگو را به صورت ترکیبی استفاده کرد: از Command برای عملیات‌های محاسبه‌پذیر و سبک (مثل جابه‌جایی لایه‌ها) و از Memento برای تغییرات پیچیده و بازگشت‌ناپذیر (مثل اعمال فیلترهای سنگین) بهره برد.

🚀 مطالعه آزاد: استفاده از الگوی Memento در حل یکی از بزرگترین چالش های دنیای Game

هدف از این بخش کتاب، نمایش روش حل یکی از بزرگترین چالش های بازیها و موتورهای بازی است! بله!

زمان معکوس در بازی هایی مانند Prince Of Persia: The sands of time!



💡 **توجه:** تکنیک پیش رو، در صحنه آهسته Replay بازیهای همچون Fifa نیز به کار می رود! منتها فرق Replay بازیها در تکرار جریان رو

به جلوی بازی در لحظات حساس بازی است! (بر خلاف جهت معکوس بازی هایی همچون Prince Of Persia)

یکی از جذاب ترین قابلیت هایی که در بعضی از بازی های ویدئویی دیده ایم، امکان بازگرداندن زمان رو به عقب یا Rewind Time است.

اما منظور از «بازگرداندن زمان» در یک بازی دقیقاً چیست؟ آیا باید تمام اتفاقاتی را که در چند ثانیه گذشته رخ داده اند، دوباره اجرا کنیم؟

یا باید چیزی از گذشته بازی را از قبل نگه داشته باشیم تا هنگام فشردن یک دکمه بتوانیم بازی را به وضعیت قبلی برگردانیم؟

برای پاسخ به این سؤال، یکی از بهترین مثال ها بازی معروف Prince of Persia: The Sands of Time است.

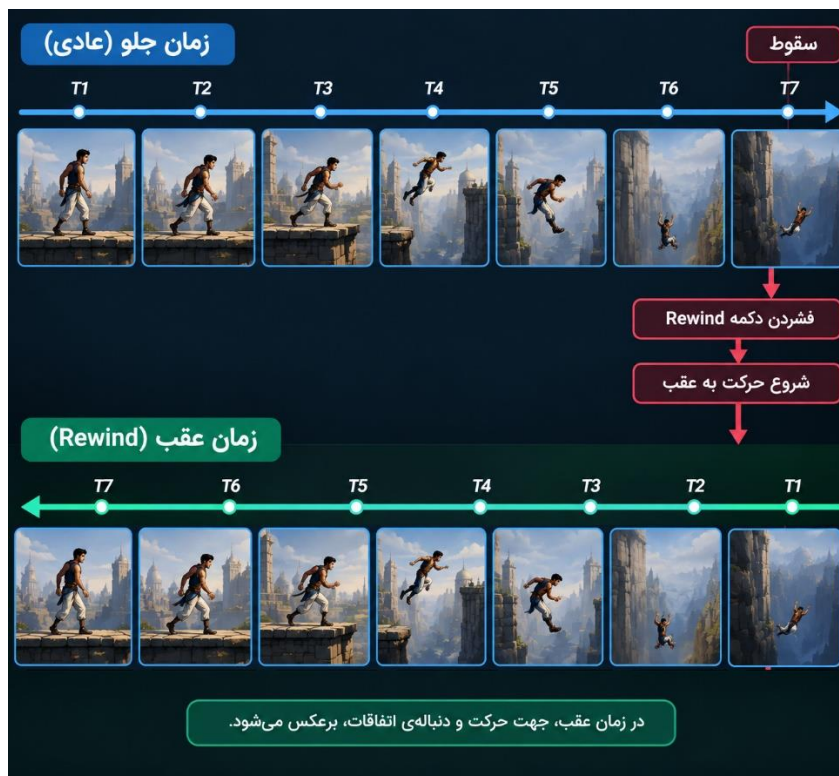
فرض کنید شخصیت اصلی بازی در حال حرکت است. او روی یک سکو می دود، می پرد و ناگهان یک اشتباه رخ می دهد!

شخصیت از لبه ی یک دره سقوط می کند. در حالت عادی، باید بازی اجازه دهد سقوط ادامه پیدا کند و احتمالاً شخصیت بمیرد.

اما بازیکن قابلیت کنترل زمان را در اختیار دارد. او دکمه ی مخصوص Rewind را فشار می دهد. در این لحظه اتفاق بسیار جالبی رخ می دهد:

حرکت بازی دیگر به سمت آینده ادامه پیدا نمی کند! در عوض، شخصیت شروع به حرکت در مسیر اتفاقات گذشته می کند.

به بیان بهتر اگر در جریان نرمال حرکت زمان رو به جلو، وقایع به صورت بخش بالای تصویر زیر اتفاق بیفتند، با زدن دکمه Rewind به صورت پایین تصویر اتفاق می افتند:



یعنی اگر بازیکن دکمه را در لحظه‌ی T7 فشار دهد، بازی باید بتواند وضعیت شخصیت را در T6، T5، T4 و ... دوباره نمایش دهد. در نتیجه، از دید بازیکن به نظر می‌رسد زمان به سمت گذشته در حرکت است! یعنی بازیکنی که در حال سقوط است، به سمت بالا و عقب حرکت می‌کند و تمام شخصیت‌های دوست و دشمن نیز به صورت معکوس در زمان حرکت می‌کنند و آتش به صوت معکوس از بالا به پایین و جریان آب آبشار از پایین به بالا حرکت می‌کند! گویی شما فیلمی را از آخر به اول مشاهده می‌کنید! اما از دید موتور بازی، اتفاق کاملاً متفاوتی در حال رخ دادن است. موتور بازی در واقع زمان را به معنای فیزیکی معکوس نمی‌کند. بلکه State یا وضعیت‌های قبلی بازی را که از قبل ذخیره شده‌اند، دوباره پیدا و روی Game Object اعمال می‌کند.

وقتی بازیکن دکمه‌ی Rewind را فشار می‌دهد، Engine باید بتواند به سؤالاتی مانند این پاسخ دهد:

شخصیت در دو ثانیه‌ی قبل کجا بوده است؟

سرعت حرکت او چقدر بوده است؟

چه جهتی داشته است؟

انیمیشن او در چه مرحله‌ای بوده است؟

وضعیت داخلی شخصیت در چه State‌ای بوده؟ آیا در حال دویدن بوده یا پریدن؟

اشیای متحرک محیط در چه وضعیتی بوده‌اند؟

اگر Physics فعال باشد، وضعیت حرکت اجسام چه بوده است؟

بنابراین Rewind Time را می‌توان به شکل دقیق‌تری این‌گونه تعریف کرد:

Rewind Time قابلیت است که به Engine اجازه می‌دهد وضعیت گذشته‌ی بازی را از روی تاریخچه‌ی ذخیره‌شده State‌ها، بازسازی و دوباره روی Object‌های بازی اعمال کند.

? مسئله این است: چگونه تاریخچه‌ی محدودی از وضعیت بازی را ذخیره کنیم و در زمان نیاز بتوانیم یک وضعیت گذشته را سریع پیدا و

Restore کنیم؟

یک مثال ساده: سقوط از دره! فرض کنید بازی با نرخ ۶۰ فریم بر ثانیه اجرا می‌شود و در هر Frame، موقعیت شخصیت تغییر می‌کند:

```
Frame 1      → Position = (10, 20, 0)
Frame 2      → Position = (10.2, 20, 0)
Frame 3      → Position = (10.4, 19.8, 0)
Frame 4      → Position = (10.6, 19.2, 0)
...
Frame 60     → Position = (15, -10, 0)
```

در Frame شماره ۶۰ شخصیت از دره سقوط کرده است. اگر بازیکن اکنون Rewind را فعال کند، Engine باید بتواند چیزی شبیه به این را انجام دهد و وضعیت شخصیت را در هر مرحله به وضعیت قبلی برگرداند!

```
Frame 60
↓
Frame 59
↓
Frame 58
↓
Frame 57
↓
...
Frame 40
```

بنابراین Engine باید تاریخچه‌ی Frame های گذشته را در اختیار داشته باشد. اما ذخیره‌ی نامحدود تاریخچه ممکن نیست.

اینجاست که Ring Buffer در کنار Memento حلال مسئله می‌شود. آیا باید تمام تاریخ بازی را ذخیره کنیم؟ خیر. معمولاً بازیکن نیازی ندارد بتواند بازی را تا چند ساعت قبل برگرداند. ممکن است طراحی بازی فقط اجازه دهد:

5 ثانیه

10 ثانیه

20 ثانیه

به عقب برگردیم. مثلاً اگر بخواهیم تمام فریم های ۱۰ ثانیه‌ی گذشته را به صورت کامل نگه داریم و بازی با نرخ ۶۰ FPS اجرا شود:

$$10 \times 60 = 600 \text{ Snapshot}$$

پس کافی است حداکثر ۶۰۰ وضعیت را نگه داریم. اگر Snapshot ها را هر ۵۰ میلی‌ثانیه ذخیره کنیم، در هر ثانیه ۲۰ عدد Snapshot خواهیم داشت. برای ۱۰ ثانیه:

$$10 \times 20 = 200 \text{ Snapshot}$$

بنابراین میزان حافظه‌ی موردنیاز را می‌توان از روی مدت Rewind و نرخ Snapshot محاسبه کرد. این همان ایده استفاده از یک تاریخچه محدود است که در سند مبنا نیز بر آن تأکید شده است.

? Snapshot چیست؟ برای حل مسئله، به جای ذخیره‌ی خود «زمان»، از وضعیت بازی در زمان‌های مختلف عکس می‌گیریم! به این

عکس Snapshot می‌گوییم! هر Snapshot تصویری از وضعیت یک Object در یک لحظه مشخص است.

```
struct GameState
{
    float timestamp;

    float posX;
    float posY;
    float posZ;

    float rotX;
    float rotY;
    float rotZ;

    float velocityX;
    float velocityY;
    float velocityZ;

    int animationState;
};
```

در این ساختار:

timestamp زمان Snapshot است. (تعداد میلی ثانیه های گذشته از شروع بازی)

posX/Y/Z موقعیت Object است.

rotX/Y/Z چرخش Object است.

velocityX/Y/Z سرعت Object است.

animationState وضعیت انیمیشن است.

پس یک Snapshot می تواند تقریباً چنین معنایی داشته باشد:

Time = 12.50

Position = (15,8,2)

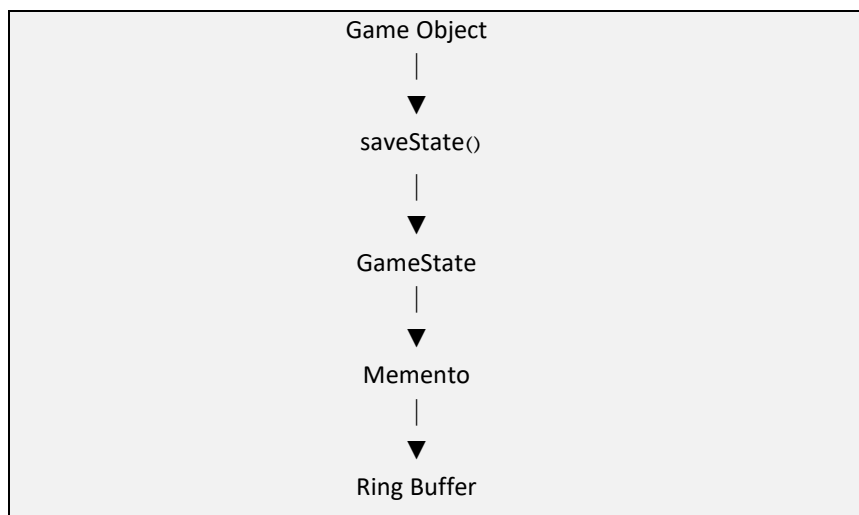
Rotation = (0,90,0)

Velocity = (2,0,0)

Animation = Running

🔗 ارتباط Snapshot با الگوی Memento : در معماری مبتنی بر Memento Pattern، چنین ساختاری می تواند نقش

Memento را داشته باشد. ایده ی Memento این است که Object بتواند وضعیت داخلی خود را در یک ساختار نگهداری کند و بعداً همان وضعیت را Restore کند.



GameState = Memento

بنابراین:

Ring Buffer نگهدارنده Memento اشیای مختلف است! خود Game Object مسئول ساخت وضعیت خودش است و سیستم تاریخچه، مسئول نگهداری Snapshot ها خواهد بود. این تفکیک باعث می شود Buffer از منطق Physics، Animation و Character مستقل بماند. چرا Ring Buffer؟ فرض کنید یک آرایه با ظرفیت ۵ داریم و وضعیت ۵ لحظه S1 تا S5 را در آن قرار داده ایم:

[S1][S2][S3][S4][S5]

وقتی Snapshot جدیدی ایجاد می شود، نمی خواهیم Buffer بزرگ تر شود. اگر Buffer پر باشد، قدیمی ترین اطلاعات باید کنار گذاشته شوند. بنابراین اگر وضعیت به این حالت رسید:

[S1][S2][S3][S4][S5]

با ثبت شدن S6:

[S6][S2][S3][S4][S5]

و سپس با آمدن S7:

[S6][S7][S3][S4][S5]

در نتیجه همیشه فقط یک پنجره محدود از تاریخچه را نگه می داریم. این همان ویژگی ای است که Ring Buffer را برای Rewind Time مناسب می کند!

💡 ظرفیت ثابت است و داده های قدیمی به صورت خودکار با داده های جدید **Overwrite** می شوند.

Ring Buffer دقیقاً چگونه کار می کند؟ برای یک Ring Buffer ساده، سه مقدار بسیار مهم داریم:

capacity: حداکثر تعداد Snapshot هایی که Buffer می تواند نگه دارد.

writeIndex: محل نوشتن Snapshot بعدی (دقت شود که محل snapshot کنونی نیست و بعدی است).

validCount: تعداد Snapshot های معتبر فعلی.

مثلاً:

capacity = 8, writeIndex = 3, validCount = 8

یعنی Buffer کاملاً پر است و Snapshot بعدی باید در خانه ی شماره ی ۳ نوشته شود. پس:

writeIndex = (writeIndex + 1) % capacity;

باعث می شود Index به شکل دایره ای حرکت کند. مثلاً:

۰ → ۱ → ۲ → ۳ → ۴ → ۵ → ۶ → ۷ → ۰ → ۱ ...

👤 تفاوت Ring Buffer معمولی با Rewind Buffer: هر Ring Buffer الزاماً برای Rewind مناسب نیست. در بسیاری از

Queue ها، وقتی داده را می خوانیم، آن داده مصرف می شود. اما در Rewind ما همیشه نمی خواهیم Snapshot را حذف کنیم. فرض کنید:

S1

S2

S3

S4

S5

بازیکن می خواهد به S3 برگردد. ما باید S3 را بخوانیم، اما بهتر است به محض خواندن، آن را از تاریخچه حذف نکنیم. بنابراین:

Buffer معمولی: Read → Consume

Rewind Buffer: Read → Peek

یعنی هنگام Rewind، Snapshot ها فقط Peek می شوند و چه در زمان رها کردن دکمه یا همزمان با Rewind، مصرف می شوند. این یکی از

مهم ترین تفاوت های معماری Rewind Time با یک Queue معمولی است. در نتیجه ما هم باید بتوانیم فقط Peek کنیم و هم Consume!

🌸 پیاده سازی عمومی Ring Buffer در C++: اکنون می توانیم خود Ring Buffer را مستقل از GameState پیاده سازی کنیم.

```
#include <cstddef>
#include <stdexcept>
#include <utility>
#include <vector>

template <typename T>
class RingBuffer{
public:
    explicit RingBuffer(std::size_t capacity)
        : buffer_(capacity),
          capacity_(capacity),
          writeIndex_(0),
          validCount_(0) {

        if (capacity == 0) {
            throw std::invalid_argument("RingBuffer capacity must be greater than zero");
        }
    }

    std::size_t size() const noexcept {
        return validCount_;
    }

    std::size_t capacity() const noexcept {
        return capacity_;
    }

    bool empty() const noexcept {
        return validCount_ == 0;
    }
};
```

```

bool full() const noexcept {
    return validCount_ == capacity_;
}

void clear() noexcept {
    writeIndex_ = 0;
    validCount_ = 0;
}

void push(const T& value) {
    buffer_[writeIndex_] = value;
    writeIndex_ = (writeIndex_ + 1) % capacity_;
    if (validCount_ < capacity_) {
        ++validCount_;
    }
}

// move version
void push(T&& value) {
    buffer_[writeIndex_] = std::move(value);
    writeIndex_ = (writeIndex_ + 1) % capacity_;
    if (validCount_ < capacity_) {
        ++validCount_;
    }
}

const T& atFromNewest( std::size_t stepsBack) const {
    if (validCount_ == 0){
        throw std::out_of_range("RingBuffer is empty");
    }
    if (stepsBack >= validCount_) {
        throw std::out_of_range("stepsBack is outside valid range");
    }
    const std::size_t index = (writeIndex_ + capacity_ -1- stepsBack)% capacity_;
    // منهای یک به این خاطر که ایندکس نوشتن روی اولین خانه بعدی قابل نوشتن قرار دارد
    // نه روی آیتم آخری که اضافه شده است، استفاده از جمع و سپس تقسیم ظرفیت کل هم برای
    // خاصیت wrap around استفاده می شود! یعنی در ایندکس منفی یا ایندکس بیش از ظرفیت
    // ظرفیت بافر، writeindex را از طرف دیگر بافر قرار دهیم تا دور بزند
    return buffer_[index];
}

private:
    std::vector<T> buffer_;
    std::size_t capacity_;
    std::size_t writeIndex_;
    std::size_t validCount_;
};

```

? چگونه Snapshot قبلی را پیدا کنیم؟ فرض کنید:

capacity = 8

writelIndex = 3

یعنی Snapshot بعدی در خانه‌ی ۳ نوشته خواهد شد. پس جدیدترین Snapshot در خانه‌ی ۲ قرار دارد. بنابراین:

stepsBack = 0 → index 2

stepsBack = 1 → index 1

stepsBack = 2 → index 0

stepsBack = 3 → index 7

stepsBack = 4 → index 6

فرمول: 

index = (writeIndex + capacity - 1 - stepsBack) % capacity;

این فرمول یکی از مهم‌ترین بخش‌های کل سیستم است. مثلاً:

writeIndex = 3

capacity = 8

stepsBack = 3

خواهیم داشت:

$$(3 + 8 - 1 - 3) \% 8 = 7 \% 8 = 7$$

پس Snapshot موردنظر در خانه‌ی ۷ قرار دارد.

 ساخت Ring Buffer مخصوص Rewind Time : اکنون Ring Buffer را با GameState ترکیب می‌کنیم.

```
struct Vec3
{
    float x = 0.0f;
    float y = 0.0f;
    float z = 0.0f;
};
struct GameState
{
    float timestamp = 0.0f;

    Vec3 position;
    Vec3 rotation;
    Vec3 velocity;

    int animationState = 0;
};
```

اکنون:

```
class TimeRewindBuffer {
public:
    explicit TimeRewindBuffer(std::size_t maxFrames)
        : buffer_(maxFrames),
          capacity_(maxFrames),
          writeIndex_(0),
          validCount_(0) {
        if (maxFrames == 0) {
            throw std::invalid_argument("maxFrames must be greater than zero");
        }
    }
};
```

```

void record(const GameState& state) {
    buffer_[writeIndex_] = state;

    writeIndex_ = (writeIndex_ + 1) % capacity_;

    if (validCount_ < capacity_) {
        ++validCount_;
    }
}

bool canRewind() const noexcept {
    return validCount_ > 1;
}

std::size_t size() const noexcept {
    return validCount_;
}

const GameState& getFromNewest(std::size_t stepsBack) const{
    if (validCount_ == 0) {
        throw std::out_of_range("No state has been recorded");
    }

    if (stepsBack >= validCount_){
        throw std::out_of_range("Requested state is unavailable");
    }

    const std::size_t index = (writeIndex_ + capacity_ - 1 - stepsBack) % capacity_;
    return buffer_[index];
}

void clear() noexcept {
    writeIndex_ = 0;
    validCount_ = 0;
}

private:
    std::vector<GameState> buffer_;
    std::size_t capacity_;
    std::size_t writeIndex_;
    std::size_t validCount_;
};

```

در اینجا هر خانه‌ی Ring Buffer یک Memento از وضعیت بازی است.

تا اینجا کار، ما می‌توانیم مثلاً ۱۰ فریم به عقب برگردیم!

stepsBack = 10

یعنی 10 تا Snapshot به عقب برو. اما بازیکن این‌گونه فکر نمی‌کند! او نمی‌گوید: لطفاً ۱۰ تا Snapshot به عقب برو!

بلکه او می‌گوید: ۲ ثانیه به عقب برگرد!

بنابراین باید Snapshot ها دارای timestamp باشند.


```
GameState state;

state.timestamp = gameTime;
state.position = player.position();
state.rotation = player.rotation();
state.velocity = player.velocity();
```

اگر Snapshot هر ۵۰ میلی‌ثانیه ذخیره شود:

```
Snapshot interval = 0.05 sec
```

پس برای هر ثانیه:

```
20 Snapshot -> 1 second
```

و برای ۱۰ ثانیه:

```
200 Snapshot
```

Rewind بر اساس زمان چگونه انجام می‌شود؟ فرض کنید:

```
float currentRewindTime = gameTime;
const float rewindSpeed = 2.0f;
```

💡 در حالت Rewind:

```
currentRewindTime -= deltaTime * rewindSpeed;
```

DeltaTime یا DT چیست؟ عدد اعشاری 1.0 تقسیم بر نرخ فریم یا frame rate (تعداد فریم‌های رندر شده در ثانیه)! برای مثال DeltaTime بازی که هر ثانیه، فقط ۲ فریم رندر می‌کند، 0.5 است! Dt به چه کاری می‌آید؟ اگر قرار باشد شیی در یک ثانیه از نقطه A به نقطه B حرکت کند، برای اینکه با انواع FrameRate‌های مختلف سرعتش کند یا تند نشود (با هر FrameRate‌ای، در یک ثانیه از A به B برسد)، میزان جا به جایی یعنی $B - A$ را (در هر فریم) تقسیم بر DeltaTime می‌کنند تا سرعت حرکت شی مستقل از FrameRate عمل کند! پس اگر FrameRate شما ۲ فریم در ثانیه است، در هر فریم شی به اندازه نصف مسیر حرکت می‌کند و اگر frame rate برابر ۳ فریم در ثانیه است، شی در هر فریم یک سوم مسیر را حرکت می‌کند! پس حاصل تقسیم $1.0 / \text{Frame Rate}$ گارانتی می‌کند شی در مدت زمان مورد نظر ما، با هر frame rate‌ای، از نقطه A به نقطه B حرکت می‌کند! پس در مثال ما اگر:

```
deltaTime = 0.016
rewindSpeed = 2
```

در هر Frame تقریباً:

```
0.032 second
```

به سمت گذشته حرکت می‌کنیم. بنابراین اگر:

```
currentRewindTime = 20.000
```

باشد، به مرور تبدیل می‌شود به:

```
19.968
19.936
19.904
19.872
...
```

این همان چیزی است که از دید بازیکن به صورت «حرکت زمان به عقب» دیده می‌شود. در واقع یک متغیر زمان هدف داریم که به سمت گذشته حرکت می‌کند.

❌ مشکل مهم Snapshot: دقیقاً در زمان موردنظر وجود ندارد! فرض کنید Snapshot های ما این‌ها هستند:

```
10.00
10.05
10.10
10.15
```

و بازیکن می‌خواهد وضعیت زیر را ببیند:

```
10.03
```

اما Snapshot دقیقاً در 10.30 وجود ندارد. پس چه کار کنیم؟

دو Snapshot اطراف زمان هدف را پیدا می‌کنیم:

```
10.00          10.05
S1 ----- S2
           ↑
        10.03
```

و سپس وضعیت بین آن‌ها را محاسبه می‌کنیم. این کار Interpolation نام دارد.

🎯 فرمول محاسبه Interpolation:

```
alpha = (targetTime - stateA.timestamp) / (stateB.timestamp - stateA.timestamp);
```

🎯 برای مثال:

```
stateA = 10.00
stateB = 10.05
target = 10.03
```

بنابراین:

```
alpha =
(10.03 - 10.00) / (10.05 - 10.00)
= 0.03 / 0.05 = 0.6
```

یعنی وضعیت هدف ۶۰ درصد مسیر بین دو Snapshot قرار دارد.

🎯 Interpolation برای Position: یک تابع ساده Linear Interpolation یا همان Lerp است!

```
Vec3 lerp(
    const Vec3& a,
    const Vec3& b,
    float t)
{
    return
    {
        a.x + (b.x - a.x) * t,
        a.y + (b.y - a.y) * t,
        a.z + (b.z - a.z) * t
    };
}
```

```

GameState interpolate(
    const GameState& a,
    const GameState& b,
    float targetTime)
{
    float alpha =
        (targetTime - a.timestamp)/(b.timestamp - a.timestamp);

    if (alpha < 0.0f)
        alpha = 0.0f;

    if (alpha > 1.0f)
        alpha = 1.0f;

    GameState result;

    result.timestamp = targetTime;

    result.position =
        lerp(
            a.position,
            b.position,
            alpha
        );

    result.velocity =
        lerp(
            a.velocity,
            b.velocity,
            alpha
        );

    result.rotation =
        lerp(
            a.rotation,
            b.rotation,
            alpha
        );

    result.animationState =
        alpha < 0.5f
        ? a.animationState
        : b.animationState;

    return result;
}

```

با این روش، Rewind دیگر محدود به Frame‌های ذخیره‌شده نیست و می‌تواند یک زمان میانی را نیز بازسازی کند.

💡 نکته‌ی مهم درباره‌ی **Rotation**: برای Position، استفاده از Lerp ساده معمولاً قابل قبول است.

اما برای Rotation سه‌بعدی، Lerp ساده زاویه‌های Euler همیشه انتخاب مناسبی نیست.

در یک Engine واقعی معمولاً Rotation به صورت:

Quaternion

ذخیره می‌شود و برای Interpolation از روش‌هایی مانند:

Slerp

استفاده می‌شود. بنابراین در یک سیستم Rewind واقعی، بهتر است به جای:

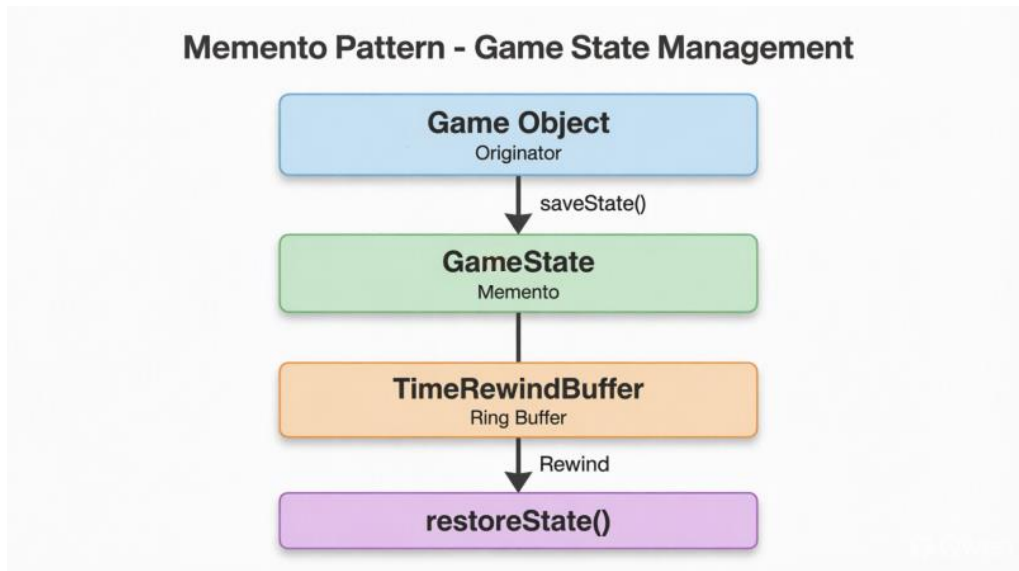
Vec3 rotation;

از ساختاری مانند:

Quaternion rotation;

استفاده شود.

 معماری کلی سیستم: اکنون می‌توانیم معماری را به شکل زیر ببینیم:



در این معماری:

Game Object

وضعیت خودش را می‌سازد.

GameState

نماینده‌ی Snapshot یا Memento است.

TimeRewindBuffer

تاریخچه را نگه می‌دارد.

RewindManager

تصمیم می‌گیرد چه زمانی Snapshot ذخیره شود و چه زمانی Rewind انجام شود.

❖ پیاده‌سازی یک Character ساده: برای اینکه مسئله کاملاً ملموس شود، یک Character ساده تعریف می‌کنیم.

```
class Player {
public:
    GameState saveState(float time) const {
        return {
            time,
            position_,
            rotation_,
            velocity_,
            animationState_
        };
    }
    void restoreState( const GameState& state) {
        position_ = state.position;
        rotation_ = state.rotation;
        velocity_ = state.velocity;
        animationState_ = state.animationState;
    }
    void update(float dt){
        position_.x += velocity_.x * dt;
        position_.y += velocity_.y * dt;
        position_.z += velocity_.z * dt;
        rotation_.y += 90.0f * dt;
        animationState_ = std::fmod( currentTime_, 1.0f ) < 0.5f ? 0 : 1;
        currentTime_ += dt;
    }
    const Vec3& position() const {
        return position_;
    }
private:
    Vec3 position_;
    Vec3 rotation_;

    Vec3 velocity_ {
        2.0f,
        0.0f,
        0.0f
    };

    int animationState_ = 0;
    float currentTime_ = 0.0f;
};
```

❖ RewindManager: حالا یک لایه‌ی بالاتر ایجاد می‌کنیم:

```
class RewindManager {
public:
    explicit RewindManager( std::size_t maxSnapshots)
        : history_(maxSnapshots){}
```

```

void record( float gameTime, const Player& player) {
    if (rewinding_)
        return;
    history_.push( player.saveState(gameTime) );
}

bool canRewind() const noexcept {
    return history_.size() > 1;
}

const GameState& rewindStep( std::size_t stepsBack) const {
    return history_.atFromNewest( stepsBack );
}

void startRewind() {
    rewinding_ = true;
}

void stopRewind() {
    rewinding_ = false;
}

bool isRewinding() const noexcept {
    return rewinding_;
}

private:

    RingBuffer<GameState> history_;

    bool rewinding_ = false;
};

```

یک نکته‌ی بسیار مهم در این کلاس وجود دارد:

```

if (rewinding_)
    return;

```

یعنی وقتی بازی در حالت Rewind است، Snapshot جدید ثبت نکن. چرا؟ چون در این حالت دیگر در حال تولید آینده نیستیم؛ بلکه در حال مشاهده‌ی گذشته هستیم.

💡 شبیه‌سازی ساده **Prince of Persia**: اکنون می‌توانیم ایده را شبیه‌سازی کنیم. در حالت عادی:

```

int main()
{
    constexpr float dt = 0.05f;

    Player player;

    RewindManager rewind(100);

    float gameTime = 0.0f;

```

```

// حالت عادی بازی
for (int frame = 0; frame < 100; ++frame) {
    gameTime += dt;
    player.update(dt);
    rewind.record(
        gameTime,
        player
    );
}

std::cout<< "Current X: "<< player.position().x << '\n';
//Rewind شروع
rewind.startRewind();

for (std::size_t step = 1; step < 40; ++step) {
    if (!rewind.canRewind())
        break;
    const GameState& past = rewind.rewindStep(step);

    player.restoreState(past);

    std::cout << "Rewind X: " << player.position().x<< '\n';
}
rewind.stopRewind();
}

```

در حالت عادی:

```

Update
↓
Save State
↓
Ring Buffer

```

در حالت Rewind:

```

Ring Buffer
↓
Find Past State
↓
Restore State
↓
Game Object

```

💡 **Rewind واقعی باید بر اساس Time باشد، نه فقط Frame**: کد قبلی برای آموزش ساده بود. در یک سیستم واقعی بهتر

است یک زمان هدف داشته باشیم:

```

float currentRewindTime = gameTime;

const float rewindSpeed = 2.0f;

```

```
while (rewinding) {
    currentRewindTime -= deltaTime * rewindSpeed;

    GameState state = getStateAtTime( currentRewindTime );
    player.restoreState(state);
}
```

اگر:

```
rewindSpeed = 2
```

باشد، زمان با سرعت دو برابر به سمت گذشته حرکت می کند. مثلاً:

20.00

19.9

19.8

19.7

19.6

...

بنابراین حرکت Rewind از نظر Engine به یک متغیر بسیار ساده تبدیل می شود:

```
rewindTime -= deltaTime * rewindSpeed;
```

 تابع **GetStateAtTime**: اکنون مسئله‌ی اصلی این است:

اگر زمان هدف مثلاً ۱۰.۳ باشد، از داخل Ring Buffer چگونه Snapshot مناسب را پیدا کنیم؟

برای یک نمونه‌ی آموزشی می‌توانیم از جست‌وجوی خطی استفاده کنیم.

```
class TimeHistory {
public:
    explicit TimeHistory(std::size_t capacity)
        : history_(capacity){}

    void record( const GameState& state){
        history_.push(state);
    }

    GameState getStateAtTime( float targetTime) const {
        if (history_.empty()) {
            throw std::runtime_error("History is empty");
        }

        const GameState& newest = history_.atFromNewest(0);
        const GameState* newer = &newest;

        const GameState* older = &newest;

        for (std::size_t step = 1; step < history_.size(); ++step){
            const GameState& candidate = history_.atFromNewest(step);
```



```

        if (candidate.timestamp <= targetTime) {
            older = &candidate;
            break;
        }
        newer = &candidate;
    }

    if (newer->timestamp == older->timestamp){
        return *newer;
    }

    float alpha = (targetTime - older->timestamp)/(newer->timestamp- older->timestamp);

    alpha = std::max(0.0f, std::min(1.0f, alpha));

    GameState result;

    result.timestamp = targetTime;

    result.position = lerp(older->position, newer->position, alpha );

    result.velocity = lerp(older->velocity, newer->velocity, alpha );

    result.rotation = lerp(older->rotation, newer->rotation, alpha);

    result.animationState =
        alpha < 0.5f ? older->animationState : newer->animationState;
    return result;
}

private:
    RingBuffer<GameState> history_;
};

```

در این پیاده‌سازی باید هدر زیر نیز وجود داشته باشد:

#include <algorithm>

? چرا هنگام Rewind نباید Record کنیم؟ این نکته از مهم‌ترین نکات طراحی سیستم است. فرض کنید تاریخچه این است:

A — B — C — D — E — F

بازیکن به F رسیده است و Rewind را فعال می‌کند. او به C برمی‌گردد:

A — B — C

اکنون اگر از C به بعد بازی را ادامه دهد، آینده‌ی قبلی دیگر معتبر نیست. یعنی:

D — E — F

دیگر آینده‌ی جدید بازی نیست. بازی ممکن است از C وارد مسیر کاملاً متفاوتی شود:

A — B — C — X — Y — Z

بنابراین Snapshot های آینده باید به تدریج با State های جدید Overwrite شوند

:Forward

Update
↓
Record
↓
Ring Buffer

اما :Rewind

Rewind Time
↓
Find State
↓
Restore State

و Rewind نباید توام با رکورد صحنه باشد:

Update
↓
Record

? وقتی بازیکن دکمه را رها می کند چه اتفاقی می افتد؟ این قسمت دقیقاً همان چیزی است که در تجربه ی بازی مهم است.

فرض کنید:

A – B – C – D – E – F

بازیکن تا F رفته است. سپس Rewind می کند:

F → E → D → C

و در C دکمه را رها می کند. از اینجا به بعد بازی باید از C ادامه پیدا کند:

A – B – C – X – Y – Z

نه اینکه دوباره به D، E و F قبلی برگردد. بنابراین آینده ی قبلی دیگر معتبر نیست و State های جدید به تدریج آن را Overwrite می کنند. این رفتار برای Rewind Time بسیار منطقی است:

بازگشت به گذشته یعنی تغییر نقطه ی ادامه ی بازی؛ بنابراین آینده ی قبلی دیگر الزاماً آینده ی معتبر نیست.

آیا فقط Position را ذخیره کنیم؟ در نگاه اول شاید فکر کنیم کافی است position را ذخیره کنیم. اما در یک بازی واقعی این کافی نیست. فرض کنید شخصیت هنگام سقوط:

Position = (10, 20, 0)

Velocity = (0, -15, 0)

داشته است. اگر فقط Position را Restore کنیم اما Velocity را Restore نکنیم، Physics Engine ممکن است بعد از Restore، حرکت اشتباهی ایجاد کند. بنابراین State می تواند شامل مواردی مانند موارد زیر باشد:

State – Physics – Time – Animation – State – Animation – Velocity – Rotation – Position

برای یک Character پیچیده‌تر حتی می‌توان داشت:

```
struct CharacterState {
    float timestamp;
    Vec3 position;
    Quaternion rotation;
    Vec3 linearVelocity;
    Vec3 angularVelocity;
    int animationId;
    float animationTime;
};
```

در Engine‌های واقعی، وضعیت Physics ممکن است بسیار پیچیده‌تر از یک Velocity ساده باشد.

💡 Rewind فقط برای Player نیست:

تا اینجا فقط Player را Rewind کردیم. اما فرض کنید شخصیت در بازی با یک جعبه برخورد کرده است.

در این صورت اگر فقط Player را به گذشته برگردانیم، ممکن است وضعیت بازی ناسازگار شود. مثلاً:

Player
Enemy
Box
Door
Projectile
Physics Body
Timer

همه ممکن است بخشی از تاریخچه باشند. یک روش این است که برای هر Object تاریخچه‌ی جداگانه داشته باشیم:

```
Player
↓
RingBuffer<PlayerState>

Enemy
↓
RingBuffer<EnemyState>

Box
↓
RingBuffer<BoxState>

Door
↓
RingBuffer<DoorState>
```

روش دیگر ذخیره کل World در یک Snapshot است:

```
struct WorldState {
    PlayerState player;

    EnemyState enemies[MAX_ENEMIES];
    BoxState boxes[MAX_BOXES];
    DoorState doors[MAX_DOORS];
};
```

انتخاب بین این دو روش به معماری Engine، تعداد Objectها، وابستگی آنها و هزینه‌ی Snapshot بستگی دارد. برای یک نمونه‌ی آموزشی، Ring Buffer جداگانه برای هر Object معمولاً ساده‌تر است.

Snapshot با چه فاصله‌ای ذخیره شود؟

یکی از تصمیم‌های مهم طراحی، نرخ Snapshot است. اگر در هر فریم، Frame Snapshot ذخیره کنیم:

FPS = 60 (Record every frame)

کیفیت Rewind بالا خواهد بود، اما حافظه‌ی بیشتری مصرف می‌شود. اگر مثلاً هر سه Frame یک Snapshot بگیریم مصرف حافظه کمتر می‌شود، اما فاصله‌ی بین Snapshotها بیشتر است. در این حالت:

Snapshot A
↓
Interpolation
↓
Snapshot B

می‌تواند Frameهای بین آنها را بازسازی کند. بنابراین یک Trade-off داریم:

Snapshot بیشتر
↓
کیفیت بهتر
↓
مصرف حافظه بیشتر

در مقابل:

Snapshot کمتر
↓
مصرف حافظه کمتر
↓
نیاز بیشتر به Interpolation

سند مبنا نیز ثبت Snapshot با نرخ کمتر از نرخ کامل Frame را به عنوان یک راه کاهش مصرف RAM مطرح می‌کند.

 **Overwrite شدن قدیمی‌ترین State:** یکی از زیباترین ویژگی‌های Ring Buffer این است که برای حذف داده‌ی قدیمی به عملیات پیچیده‌ای نیاز نداریم. فرض کنید:

capacity = 5

و داریم:

[S1][S2][S3][S4][S5]

Snapshot جدید:

S6

ثبت می‌شود:

[S6][S2][S3][S4][S5]

و سپس:

S7

می‌آید:

[S6][S7][S3][S4][S5]

در نتیجه تاریخچه همیشه یک Sliding Window روی زمان است. اگر ظرفیت Buffer معادل ۱۰ ثانیه باشد، بازی تقریباً همیشه آخرین ۱۰ ثانیه‌ی تاریخچه را نگه می‌دارد.

یک سناریوی کامل از ابتدا تا انتها: اکنون کل فرآیند را در قالب همان مثال Prince of Persia ببینیم.

✓ مرحله‌ی اول، بازی عادی: بازیکن در حال حرکت است.

```
Update
↓
GameState
↓
Ring Buffer
```

Snapshotها ذخیره می‌شوند:

[S1][S2][S3][S4][S5]...

✓ مرحله‌ی دوم، بازیکن سقوط می‌کند: فرض کنید بازیکن به یک دره رسیده است.

S1 — S2 — S3 — S4 — S5 — S6

↓
سقوط

✓ مرحله‌ی سوم، بازیکن دکمه‌ی Rewind را فشار می‌دهد: بازی وارد حالت:

Rewinding = true

می‌شود. دیگر:

Update + Record

انجام نمی‌شود. بلکه:

```
Rewind Time
↓
Find Past State
↓
Restore
```

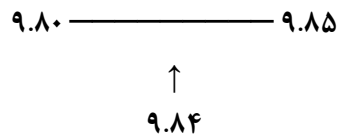
✓ مرحله‌ی چهارم، زمان به عقب حرکت می‌کند: مثلاً:

```
10.00
↓
9.96
↓
9.92
↓
9.88
↓
9.84
```

✓ مرحله‌ی پنجم، پیدا کردن Snapshotهای اطراف: مثلاً هدف:

9.84

است و Snapshot ها:



هستند. پس:

Interpolation

انجام می شود.

✓ **مرحله ی ششم، Restore :** وضعیت جدید روی Player اعمال می شود:

```
player.restoreState (pastState) ;
```

✓ **مرحله ی هفتم، بازیکن دکمه را رها می کند:** فرض کنید به:

Time = 8.50

برگشته ایم. بازی از همین جا ادامه پیدا می کند.

Old Future:

80.50 - 80.60 - 80.70 - 80.80 - 9.0

New Future:

80.50 - 80.60 - 80.70 - X - Y - Z

آینده ی قبلی دیگر معتبر نیست.

✓ **تفاوت Rewind Time با Time Machine واقعی:** این نکته باید کاملاً روشن باشد.

Rewind Time در یک Game Engine به این معنی نیست که Engine می تواند واقعاً زمان را به عقب ببرد. بلکه:

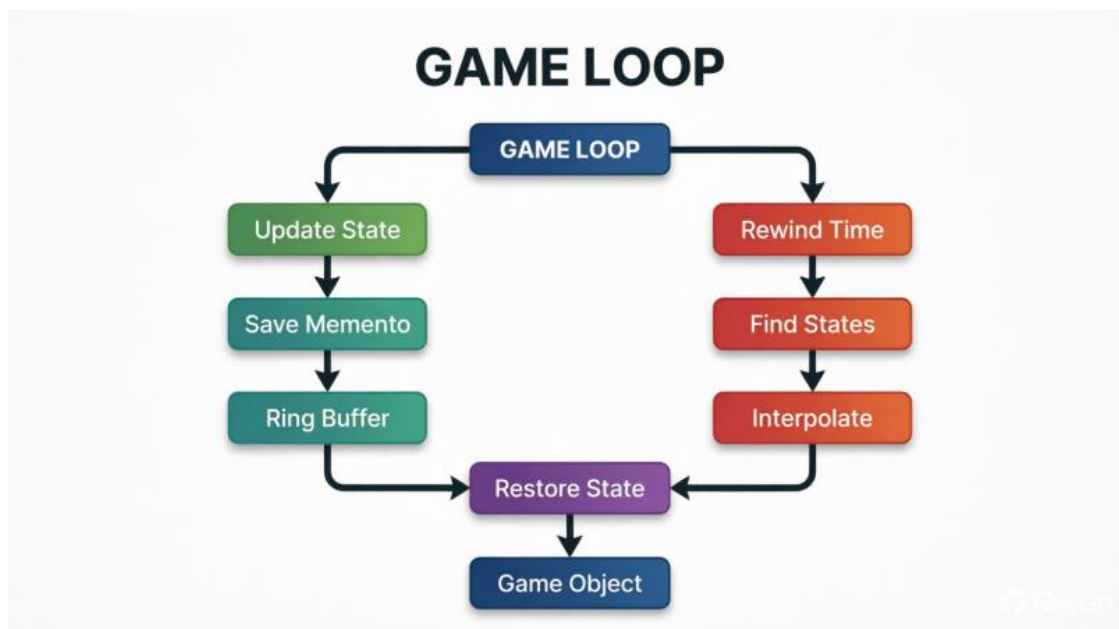
```
Past State
↓
Stored History
↓
Select State
↓
Restore State
```

پس اگر بخواهیم Rewind را در یک جمله تعریف کنیم:

✕ **Rewind Time در Engine یعنی بازسازی گذشته، نه معکوس کردن واقعی زمان.**

و این دقیقاً همان چیزی است که مثال Prince of Persia را به یک مسئله ی جالب مهندسی تبدیل می کند.

تصویر نهایی معماری: می‌توان کل سیستم را به شکل زیر خلاصه کرد: 



🕒 زنجیره‌ی مفهومی کل سیستم: اگر بخواهیم کل معماری را در چند مفهوم خلاصه کنیم:



این زنجیره در واقع تمام ایده‌ی Rewind Time را توضیح می‌دهد.

🕒 پیچیدگی زمانی: یکی از مزایای Ring Buffer این است که عملیات اصلی آن بسیار سریع هستند.

🔧 عملیات Push:

$\text{push} \rightarrow O(1)$

دسترسی به Snapshot با تعداد گام مشخص:

$\text{fromNewest}(\text{stepsBack}) \rightarrow O(1)$

چون Index با یک فرمول مستقیم محاسبه می‌شود. اما نسخه‌ی ساده‌ی:

$\text{getStateAtTime}()$

ممکن است برای پیدا کردن Snapshot های اطراف زمان هدف کل Buffer را جست‌وجو کند. بنابراین در بدترین حالت:

$\text{getStateAtTime} \rightarrow O(n)$

برای Buffer های کوچک چند صدتایی، این روش ساده و قابل قبول است. اما در یک Engine بزرگ، اگر تعداد Snapshot ها یا Object ها بسیار زیاد باشد، می توان روش های سریع تری برای جست و جوی بازه ی زمانی طراحی کرد.

 یک Optimization کوچک اما حرفه ای: برای Wrap کردن Index معمولاً از:

index = (index + 1) % capacity;

استفاده می کنیم. اما اگر ظرفیت همیشه توان عدد ۲ باشد، مثلاً:

256

512

1024

می توان از Bitwise AND استفاده کرد:

```
index = (index + 1) & (capacity - 1);
```

مثلاً:

capacity = 512

در این حالت:

capacity - 1 = 511

و می توان:

writeIndex = (writeIndex + 1) & (capacity - 1);

را به کار برد.

البته این Optimization فقط زمانی معتبر است که capacity توان عدد ۲ باشد.

همچنین در C++ مدرن نباید صرفاً با این تصور که % حتماً گلوگاه Performance است، کد را پیچیده کنیم.

ابتدا باید Performance را اندازه گیری کرد. در بسیاری از موارد، کاهش حجم Snapshot و طراحی مناسب حافظه، اهمیت بیشتری دارد.

✗ **خطاهای رایج در پیاده سازی:** در یک سیستم Rewind چند اشتباه بسیار رایج وجود دارد.

○ **ظرفیت صفر:** نباید اجازه دهیم: capacity == 0 باشد.

✗ **اشتباه گرفتن writeIndex:** writeIndex معمولاً محل نوشتن بعدی است، نه محل جدیدترین عنصر.

✗ **نادیده گرفتن validCount:** قبل از پر شدن Buffer، همه ی خانه های آرایه معتبر نیستند.

✗ **Consume کردن Snapshot هنگام Rewind:** Rewind باید بتواند Snapshot را Peek کند. در صورت نیاز آنرا

Consume کند.

✗ **ادامه دادن Record در حالت Rewind:** در هنگام Rewind نباید تاریخچه را با Snapshot های جدید مخلوط کنیم.

✗ **ذخیره ی بیش از حد State:** هر داده ای که قرار نیست Restore شود، لزوماً نباید وارد Snapshot شود.

✗ **ذخیره ی فقط Position:** اگر Physics و Animation داریم، Position به تنهایی کافی نیست.

✗ **استفاده ی ساده از Euler برای Rotation:** در سیستم های سه بعدی واقعی بهتر است از Quaternion + Slerp استفاده شود.

این نکات نیز در سند اصلی به عنوان خطاها و ملاحظات مهم پیاده‌سازی، مطرح شده‌اند.

در نهایت پس از Peek کردن stateهای گذشته اشیای بازی، می‌توانیم تابع دیگری را به صورت جداگانه برای Consume کردن stateهای مصرف شده بنویسیم تا جدای از متد دست یافتن به Stateهای گذشته، متغیرهای write index و valid count را کاهش دهیم! این بخش برای override کردن آینده جدید جای آینده قبلی و همچنین نمایش میزان بافر باقیمانده از زمان معکوس لازم است! اینکه مساله دریافت state را به متد atFromNewest بسپاریم و کار Consume کردن خانه‌های بافر را به این تابع محول کنیم، از نظر استاندارد CQS نیز بهتر است! (در فصول بعدی با آن آشنا خواهید شد)

```
void consumeFromNewest(std::size_t stepsBack) {
    if (validCount_ == 0) {
        throw std::out_of_range("No state has been recorded");
    }
    if (stepsBack > validCount_) {
        throw std::out_of_range("Requested state is unavailable");
    }
    // Move write position backwards by stepsBack states.
    writeIndex_ = (writeIndex_ + capacity_ - stepsBack) % capacity_;
    // States after the new write position are no longer valid.
    validCount_ -= stepsBack;
}
```

نتیجه‌گیری

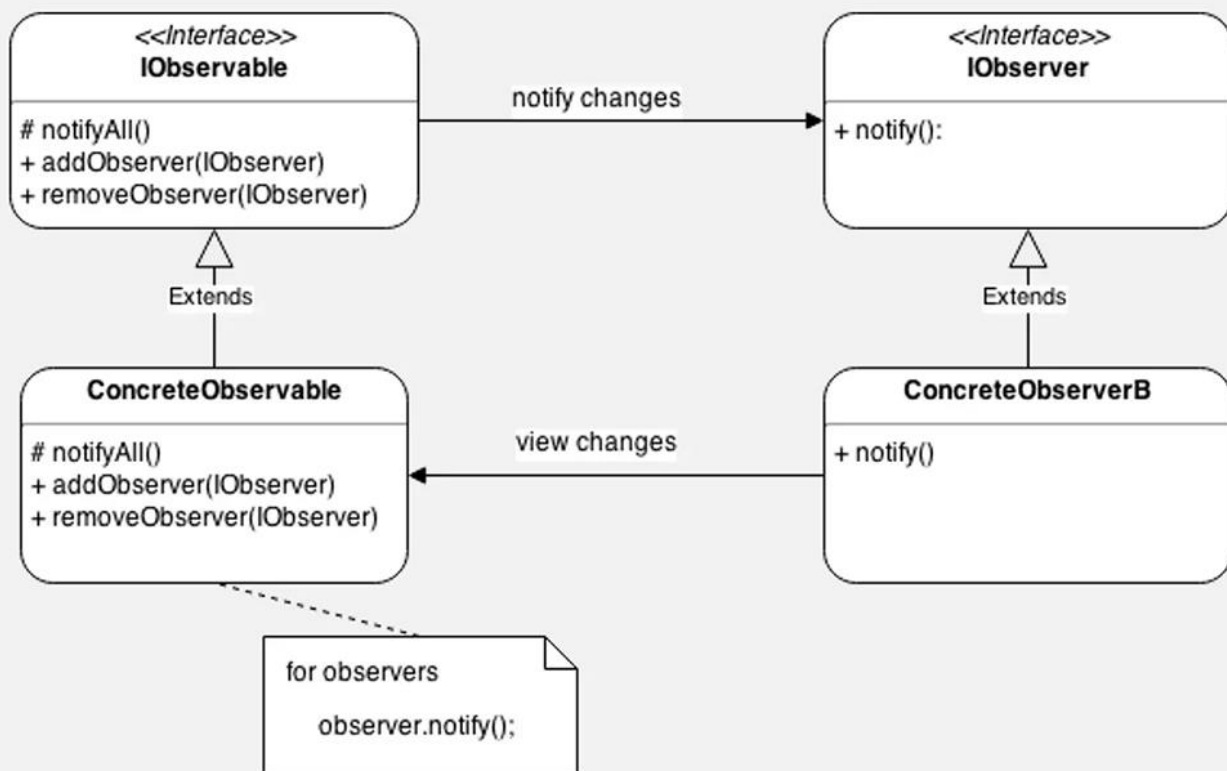
الگوی Memento یکی از ساده‌ترین و در عین حال کاربردی‌ترین الگوهای رفتاری است.

این الگو به ما امکان می‌دهد که تاریخچه‌ی تغییرات را بدون نقض اصول طراحی شیء‌گرا نگهداری کنیم.

در مثال‌های این قسمت دیدیم که چگونه می‌توان با سه کلاس ساده (Editor, EditorState, History)، یک سیستم Undo برای ویرایشگر ساخت.



Observer pattern – Class diagram



این پترن شبیه به Event Handling در Js یا سیگنال و اسلات در Qt است! به بیان بهتر وقتی شما می خواهید با اجرای یک رویداد یا Event مجموعه ای از توابع و اتفاقات را صدا بزنید، از این دیزاین پترن استفاده می کنید.

👉 در این دیزاین پترن، ما با دو نوع آجکت سر و کار داریم. Observer و Subject

Subject ⚡ به آن Publisher یا Observable هم گفته می شود. عبارت است از اشیایی که بعد از انجام کاری یا وقوع اتفاقی، یک نوتیفیکیشن را به مجموعه ای از Observer ها ارسال می کند تا آنها هر کدام کاری را انجام دهند.

Observer👁️ به آن Subscriber هم گفته می شود و عبارت است از اشیایی که بعد از انتشار یک نوتیفیکیشن از Subject ها، کاری را انجام می دهند. در واقع Observer ها گوش به فرمان Subject ها هستند تا کاری را انجام دهند! معنی لغوی Observer یعنی ناظر و در اینجا یعنی گوش به فرمان!

```
class IObserver {
public:
    virtual ~IObserver() = default;
    virtual void OnNotify() = 0;
};

class Watcher : public IObserver {
public:
    explicit Watcher(std::string name)
        : _name(name) {}

    void OnNotify() override {
        cout << "this is object Of Watcher and my name is "
              << _name << " , ";
    }

private:
    std::string _name;
};

class ISubject {
public:
    virtual ~ISubject() = default;

    virtual void AddObserver(IObserver* observer) = 0;
    virtual void RemoveObserver(IObserver* observer) = 0;
    virtual void Notify() = 0;
};

class Subject : public ISubject {
public:
    ~Subject() override = default;

    void AddObserver(IObserver* observer) override {
        _observers_list.push_front(observer);
    }

    void RemoveObserver(IObserver* observer) override {
        _observers_list.remove(observer);
    }
};
```

```

void Notify() override {
    for (auto& obj : _observers_list) {
        obj->OnNotify();
    }
}

private:
    std::forward_list<IObserver*> _observers_list;
};

class MySubject : public Subject {
};

int main() {
    MySubject my_subject_obj;

    Watcher watcher_a("Watcher A");
    Watcher watcher_b("Watcher B");
    Watcher watcher_c("Watcher C");

    my_subject_obj.AddObserver(&watcher_a);
    my_subject_obj.AddObserver(&watcher_b);
    my_subject_obj.AddObserver(&watcher_c);

    my_subject_obj.Notify();
    //this is object Of Watcher and my name is Watcher C ,
    //this is object Of Watcher and my name is Watcher B ,
    //this is object Of Watcher and my name is Watcher A ,

    my_subject_obj.RemoveObserver(&watcher_c);

    my_subject_obj.Notify();
    //this is object Of Watcher and my name is Watcher B ,
    //this is object Of Watcher and my name is Watcher A ,
}

```

🧑‍🤝‍🧑 وجه اشتراک و تفاوت الگوهای Observer و Mediator

🧑‍🤝‍🧑 **وجه اشتراک:** هر دو الگو از یک واسطه (Intermediary) استفاده می‌کنند تا ارتباط بین بخش‌های مختلف را مدیریت کنند و از وابستگی مستقیم اجزا به یکدیگر (Coupling) جلوگیری کنند.

🔗 تفاوت اصلی:

👁️ الگوی Observer (ناظر):

- ارتباط یک‌طرفه است: Subject → Observers
- وقتی وضعیت Subject تغییر می‌کند، به همه Observerهای ثبت شده اطلاع می‌دهد! (Notify)
- Observerها واکنش نشان می‌دهند اما Subject نمی‌داند چه کسی و چه کاری انجام می‌دهد! در الگوی گفته شده، Subject فقط قدرت این را دارد که بداند کدام Observerها، منتظر Notify شی Subject مدنظر هستند و چیز بیشتری از علت و کیفیت کار آنها ندارند!

🎯 مثال: Publish-Subscribe, Event Listeners

حس و حال: «اتفاقی افتاد، هر کسی خواست، واکنش نشان دهد!»

🧑‍🤝‍🧑 الگوی Mediator (میانجی):

- ارتباط مرکزی‌سازی شده است: همه با Mediator حرف می‌زنند!
- Mediator تصمیم می‌گیرد چه کسی، چه زمانی و چگونه با بخش دیگر ارتباط برقرار کند!
- منطق هماهنگی و orchestration در خود Mediator وجود دارد!

- هر بخش می داند بخشی وجود دارد که درخواست مد نظر ما را انجام می دهد، ولی نمی داند کیست! لذا درخواست مدنظر را به Mediator می دهد تا پیام را به بخش درست ارسال کند تا آن کار انجام شود!

مثال: Message Broker, Event Bus, Command Bus

- **حس و حال:** «من (Mediator) هماهنگ کننده هستم و می گویم چه کسی با چه کسی کار داشته باشد!»

مثال عملی:

- **در Observer:** وقتی Order ثبت شد، OrderCreated Event منتشر می شود. هر سرویسی که Subscribe کرده (Email, Inventory, Accounting) واکنش خودش را نشان می دهد. در نتیجه پس از ثبت سفارش، رویداد OrderCreated منتشر می شود و صادرکننده رویداد نمی داند (و برایش مهم نیست) که آیا بخش انبار موجودی را کم می کند، بخش ایمیل پیامک می فرستد یا بخش حسابداری فاکتور صادر می کند یا نه!
- **در Mediator:** OrderService به Mediator می گوید «سفارش جدید». Mediator تصمیم می گیرد اول Inventory را چک کند، سپس Payment را پردازش کند، و در نهایت Email بفرستد.

مهم ترین تفاوت (Mediator) Command و (Observer) Event: شما با Command از نقطه به واسطه Mediator به نقطه ای دیگر دستور ارسال می کنید که فلان کار را انجام بده یا فلان سرویس را بده و آن قسمت هم موظف است در صورت صحت درخواست، آن کار را انجام دهد و آن سرویس را ارائه دهد! اما Event یا رویداد به اتفاقاتی که در گذشته انجام شده اشاره می کند و مفهوم از آن این است که Command یا واقعه ای اجرا شده و اگر پس از این اتفاق، قرار است به صورت زنجیره وار، عملیات دیگری نیز اجرا شود، پیام مرا دریافت کنید و کارتان را در ادامه شروع کنید! در این ساختار همچنان که ما می توانیم Commandهایی را که مشتق شده از ICommand هستند را به واسطه Mediator ارسال کنیم و بر اساس نوع Command، پارامترهای مورد نیاز را هم به واسطه آن ارسال کنیم، Eventهایی را هم که مشتق شده از IEvent هستند را با داشتن پارامترهای خبری مورد نیاز نظیر ID یا Name یا متغیرهای دیگر، ارسال کنند!

Event Manager عمومی با الگوی Publish/Subscriber برای انواع داده های مختلف:

یکی از الگوهای زیبایی که (شاید ان شاء الله) در آینده در این کتاب به آن خواهیم پرداخت، الگوی Publish/Subscribe است که از انواع Messaging/Communication Pattern محسوب می شود! اما به دلیل ارتباط تنگاتنگ این الگو با Observer در مدیریت رویدادها، در این بخش با کمی فاصله گرفتن از الگوی Observer، به روش مدیریت Eventها یا رویدادها با استفاده از Publish/Subscribe می پردازیم.

در قطعه کد زیر، می توانیم Eventهای مختلفی برای انواع مختلف تعریف کنیم و با استفاده از یک Event Manager، هر نوع Event را به Listenerهای مختص همان نوع ارسال کنیم. در این روش، می توان برای هر نوع Event، یک یا چند تابع عمومی با ورودیهای مورد نیاز تعریف کرد تا پس از Emit شدن رویداد، Listenerهای مربوطه اجرا شوند. سپس بر اساس اطلاعات موجود در Event، مانند ID یا Name، می توان همه اشیای مرتبط، گروهی از آنها یا حتی یک شیء خاص را هدف قرار داد.

به بیان ساده، شما برای هر نوع رویداد، یک یا چند Listener تعریف می کنید و هنگام Emit کردن Event، اطلاعات مورد نیاز را همراه آن ارسال می کنید! Event Manager نیز رویداد را به Listenerهای همان نوع می رساند و آنها بر اساس اطلاعات Event، واکنش مناسب را انجام می دهند. علت اینکه در این بخش از کتاب، گریزی به این الگو زدیم نیز به این جهت است که در بخش معرفی معماری ECS نیز از همین روش برای مدیریت برخی رویدادها استفاده خواهیم کرد.

```
#include <functional>
#include <iostream>
#include <typeindex>
#include <unordered_map>
#include <vector>
```

```

class EventManager {
    using Listener = std::function<void(const void*)>;
    std::unordered_map<std::type_index, std::vector<Listener>>> m_listeners;

public:
    template<typename T>
    void on(std::function<void(const T*)> listener) {
        m_listeners[typeid(T)].push_back([listener](const void* event) {
            listener(*static_cast<const T*>(event));
        });
    }

    template<typename T>
    void emit(const T& event) {
        auto it = m_listeners.find(typeid(T));
        if (it != m_listeners.end())
            for (auto& listener : it->second)
                listener(&event);
    }
};

struct DamageEvent {
    int target;
    float amount;
};

int main() {
    EventManager eventManager;

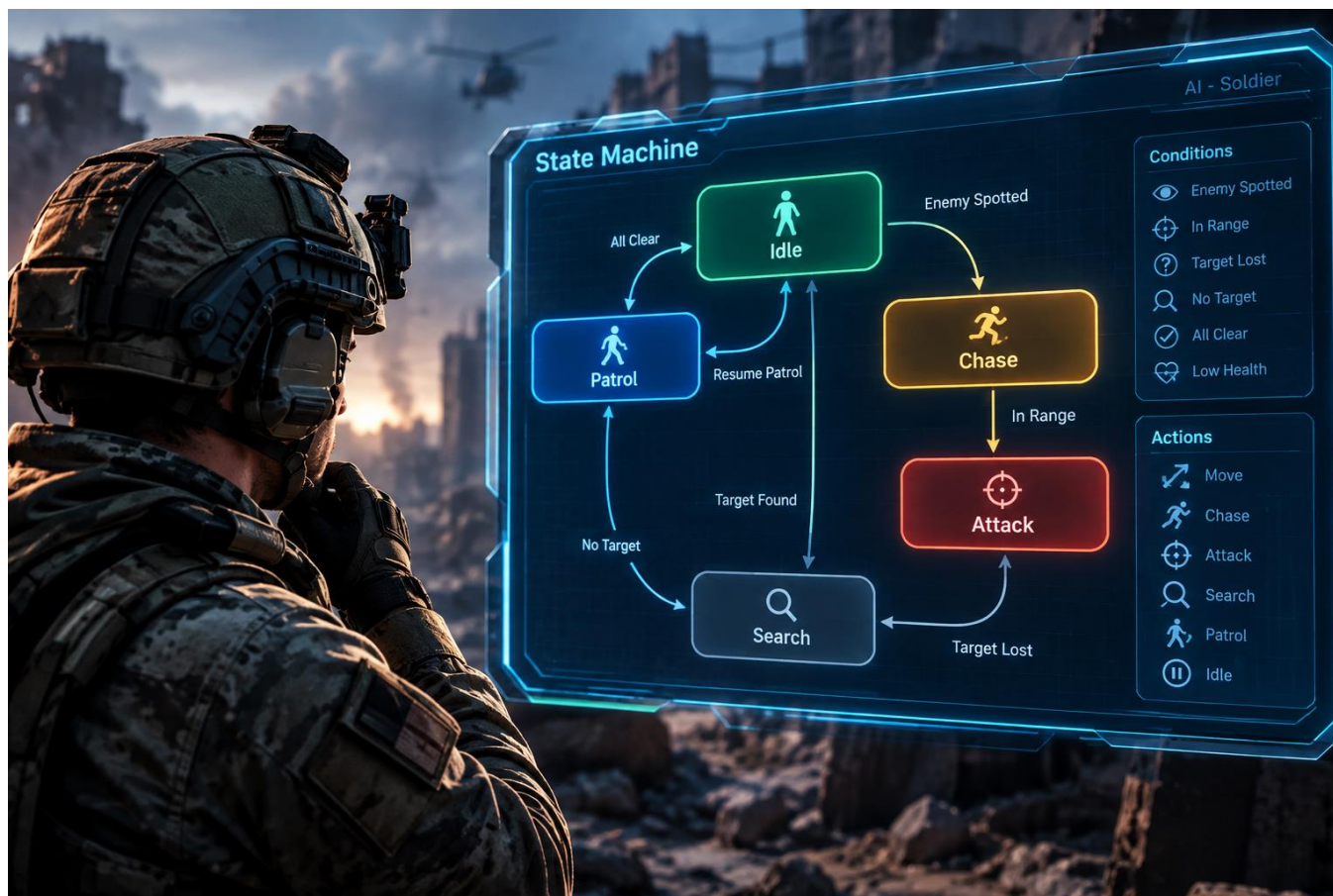
    eventManager.on<DamageEvent>([](const DamageEvent& e) {
        std::cout << "Target: " << e.target << ", Damage: " << e.amount << '\n';

        auto player = getPlayerById(event.target);
        if (player) {
            player->health -= e.amount;
            if (player->health <= 0)
                player->die();
        }
    });

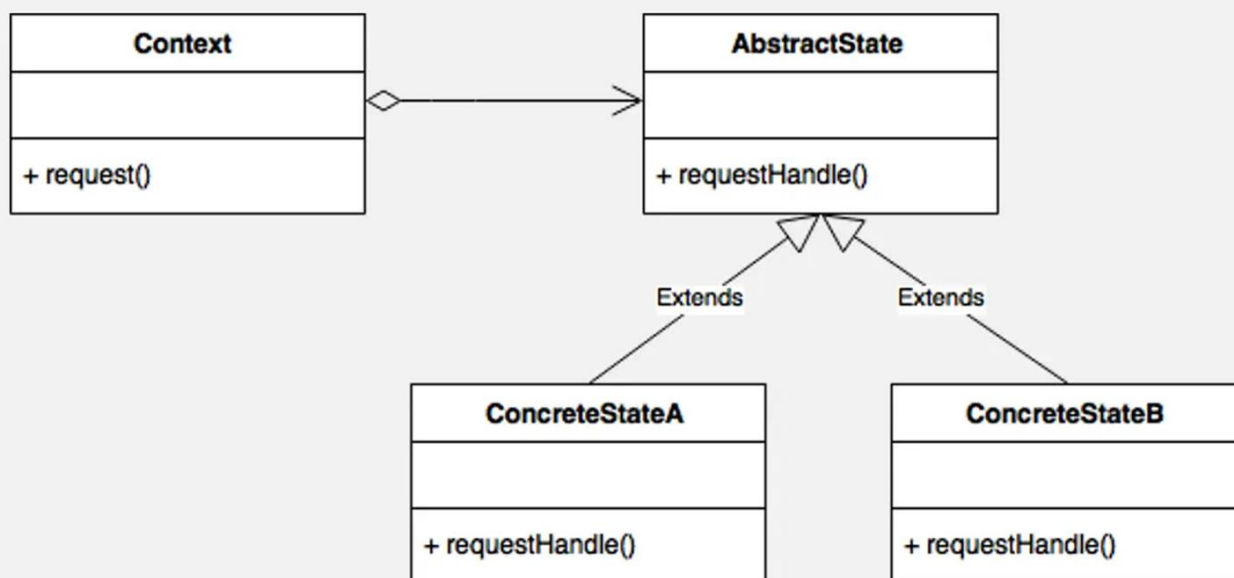
    eventManager.emit(DamageEvent{42, 25.0f});
}

```


State: پترنی از نوع Behavioral است!



State pattern – Class diagram



🔑 کارآیی: شما می خواهید در یک StateMachine از if و else if و else و switch های طولانی جلوگیری کنید!

فرض کنید یک کانکتور http دارید که وضعیت های closed و listening و connected را در خود دارد و متد open برای هر کدام از این وضعیت ها، رفتار خاص خود را خواهد داشت!

```
#include <iostream>
#include <memory>

class Context; // Forward declaration of Context
// Base State interface
class State {
public:
    virtual ~State() = default;
    virtual void open(Context* ctx) = 0;
    virtual void close(Context* ctx) = 0;
};
// Forward declarations of concrete states
class ClosedState;
class ListeningState;
class ConnectedState;
// Context owns the current state
class Context {
    std::unique_ptr<State> state;

public:
    Context(std::unique_ptr<State> initial) : state(std::move(initial)) {}

    void setState(std::unique_ptr<State> newState) { state = std::move(newState); }
    void open() { state->open(this); }
    void close() { state->close(this); }
};
// Concrete State: Closed
class ClosedState : public State {
public:
    void open(Context* ctx) override;
    void close(Context*) override { std::cout << "Already closed.\n"; }
};
// Concrete State: Listening
class ListeningState : public State {
public:
    void open(Context* ctx) override;
    void close(Context* ctx) override;
};
// Concrete State: Connected
class ConnectedState : public State {
public:
    void open(Context*) override { std::cout << "Already connected.\n"; }
    void close(Context* ctx) override;
};
// Closed -> Listening
void ClosedState::open(Context* ctx) {
    std::cout << "Opening connection...\n";
    ctx->setState(std::make_unique<ListeningState>());
}
```



```
// Listening -> Connected
void ListeningState::open(Context* ctx) {
    std::cout << "Connecting...\n";
    ctx->setState(std::make_unique<ConnectedState>());
}

void ListeningState::close(Context* ctx) {
    std::cout << "Closing connection...\n";
    ctx->setState(std::make_unique<ClosedState>());
}

// Connected -> Closed
void ConnectedState::close(Context* ctx) {
    std::cout << "Disconnecting...\n";
    ctx->setState(std::make_unique<ClosedState>());
}

int main() {
    Context ctx(std::make_unique<ClosedState>());

    ctx.open(); // Closed -> Listening
    ctx.open(); // Listening -> Connected
    ctx.open(); // Already Connected
    ctx.close(); // Connected -> Closed
    ctx.close(); // Already Closed
}
```

در این الگو، ما یک سیستم State Machine داریم که شامل **N** حالت (State) مختلف و **M** متد کلیدی است که می‌توانند وضعیت ماشین حالت را تغییر دهند. ساختار و رفتار این الگو به شرح زیر است:

🧠 **کلاس Context (بستر اصلی):** این کلاس بدنه اصلی State Machine است. درون Context، یک اشاره‌گر (Pointer) به عنوان نماینده وضعیت کنونی سیستم وجود دارد. با تزریق InitialState به این آبجکت، وضعیت اولیه سیستم تعیین می‌شود.

🏠 ساختار کلاس‌های State:

- تمام **M** متد، هم در کلاس Context و هم در کلاس پایه State تعریف می‌شوند.
- به تعداد **N** وضعیت مختلف، ما **N** کلاس جدید از کلاس پایه State مشتق می‌کنیم که هر کدام نماینده یک وضعیت خاص از ماشین حالت هستند.
- متدها در کلاس پایه State به صورت **Pure Virtual** (یا Virtual همراه با Throw Exception) تعریف می‌شوند و در هر **N** کلاس مشتق‌شده، پیاده‌سازی کامل می‌گردند.

👤 مدیریت تغییر وضعیت (Transitions):

✅ **در صورت تغییر State:** متدهای تغییردهنده، یک شیء از State جدید می‌سازند و آن را به متد setState در Context پاس می‌دهند. سپس Context، State قبلی را پاک کرده و State جدید را جایگزین می‌کند. 🔄

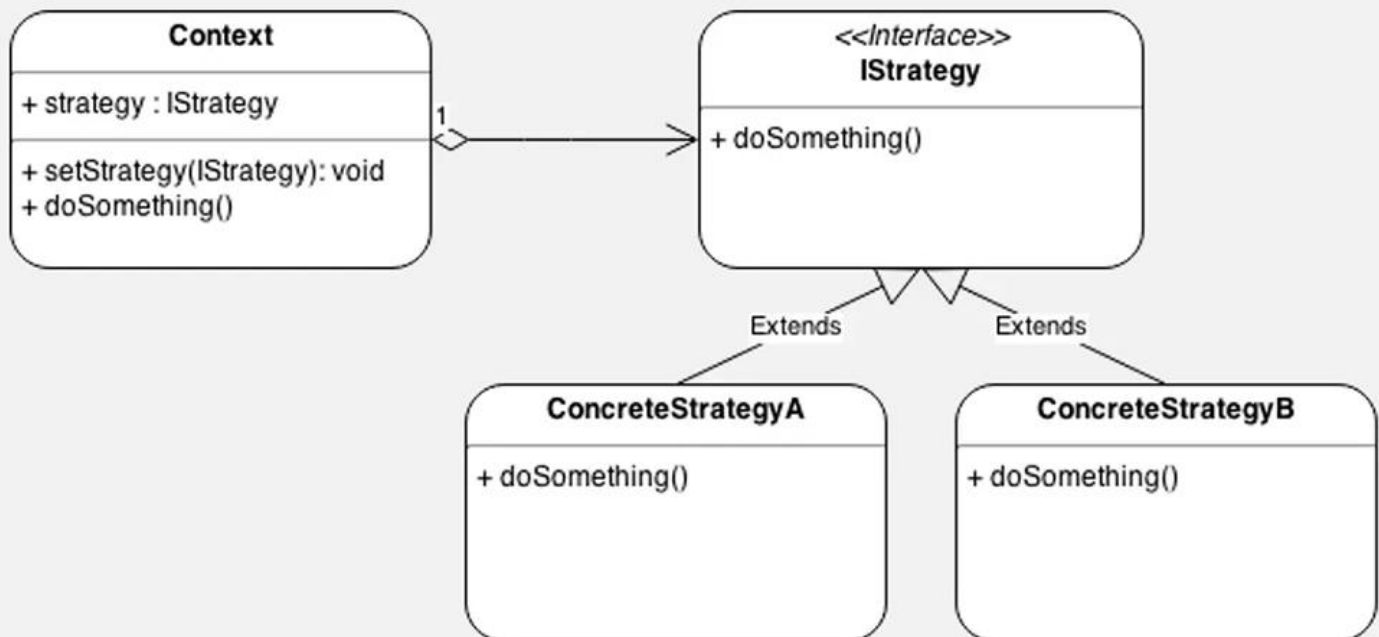
🚫 **در صورت عدم تغییر State:** اگر متدی در آن وضعیت خاص مجاز به تغییر نباشد، صرفاً یک Debug Log مبنی بر نامعتبر بودن عملیات در وضعیت فعلی ثبت می‌کند (یا Exception پرتاب می‌کند).

 **تجربه کاربر (Client Usage):** در پیاده‌سازی نهایی، کاربر فقط با شیء ساخته‌شده از Context تعامل دارد و کاملاً از State های داخلی و پیچیدگی‌های آن بی‌خبر است!

 **نکته:** حتی InitialState را هم می‌توان به صورت درونی در Constructor کلاس Context ساخت تا کاربر اصلاً درگیر آن نشود!



Strategy pattern – Class diagram



الگوی طراحی استراتژی یا **policy** الگوییست که در ران تایم، می توان به جای استفاده از یک الگوریتم مستقیم، برای انجام کاری خاص از بین الگوریتم های مختلف، مورد مناسب را انتخاب کرد و کار را با همان الگوریتم انجام داد. در واقع می توانید به صورت ران تایم یا حتی کامپایل تایم، بین انواع الگوریتم برای انجام کاری، مورد مناسب را اجرا کنید.

در نگاه اولیه برای این کار می توانیم از **branch** ها (**if** و **else** های پشت سر هم) استفاده کنیم ولی ایراد آن، در نیاز به تغییرات یک اسکریپت به کرات و نادیده گرفتن اصل **ocp** در معماری **Solid** است و لذا الگوی طراحی **strategy** سعی دارد با ارث بری کلاسی، مساله سوییچ روی الگوریتم ها را در ران تایم به گونه ای حل نماید که کلاسها برای انجام عمل گسترش و ویرایش عرضی بسته و برای انجام عمل گسترش طولی باز باشند! در مثال زیر اگر بخواهیم الگوریتم جدیدی را اضافه کنیم، می بایست همین فایل شروط را مدام عوض کنیم و این مخالف اصل **ocp** در **solid** است. ما باید تلاش کنیم جلوی ویرایش کدهای قبلی را که درست کار می کردند را بگیریم. لذا با افزودن کلاس جدید می توانیم الگوریتم های جدید را گسترش دهیم.

```
if (threshold < 16)
    insertion_sort(...);
else if (threshold >= 16 && threshold < 64)
    tim_sort (...);
else if (threshold >= 64 && threshold < 128)
    merge_sort(...);
else if ...
    ....
```

فرض کنید یک کاراکتر در یک بازی به شکل زیر داریم:

```
#include <iostream>
#include <memory>

struct Orc{
    Orc(){}

    void DoActions(){
        if(...){

        }else if(...)
        {
            ....
        }
    }
};

int main(){

    Orc orc;

    orc.DoActions();

    return 0;
}
```

متد **DoActions** را با انواع **if** ها مزین کردیم ولی هر شرط جدید و به بیان بهتر هر **Act** جدید باعث ویرایش متد **DoAction** می شود و متدی که شاید درست کار می کرده دچار چالش ویرایش می شود! اصل **OCP** تلاش دارد جلوی متدی که در گذشته کاملاً صحیح کار می کرده و تست هایش را پاس کرده را بگیرد تا نیاز به پیشرفته تر شدن تست های همان تابع نباشد!

```
// Just the interface we want to enforce, for
// different behaviors (i.e. Part of the strategy pattern)
struct ICombat{
    virtual ~ICombat(){}
    virtual void Execute() = 0;
};

struct NoCombat : public ICombat{
    void Execute(){
        std::cout << "NoCombat::Execute()\n";
    }
};

struct MeleeCombat : public ICombat{
    void Execute(){
        std::cout << "MeleeCombat::Execute()\n";
    }
};

struct Orc{
    Orc(){
        mCombatStrategy = std::make_unique<NoCombat>();
    }

    void DoActions(){
        mCombatStrategy->Execute();
    }

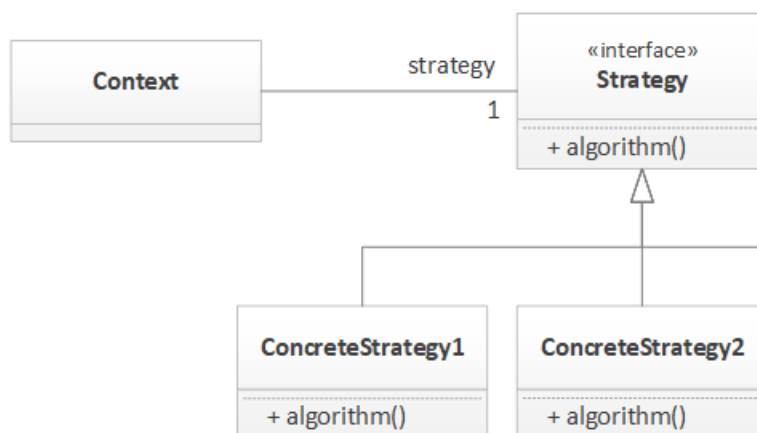
    void SetCombatStrategy(std::unique_ptr<ICombat>&& c){
        mCombatStrategy = std::move(c);
    }
private:
    std::unique_ptr<ICombat> mCombatStrategy;
};

int main(){
    Orc orc;
    orc.DoActions();
    orc.SetCombatStrategy(std::make_unique<MeleeCombat>());
    orc.DoActions();

    return 0;
}
```

اینترفیسی از **Combat** ساخته شد که نابودگر و تابع **virtual** محض **Execute** خود را دارد. سپس دو کلاس یا استراکت از آن **Implement** کردیم و در هر کدام متد **Execute** را پیاده کردیم. کلاس یا استراکت **Orc** را ساختیم که شامل اشاره گر اسمارت به اینترفیس **ICombat** است و در آن هر کدام از دو کلاس را که بخواهیم می سازیم و به اشاره گر نسبت می دهیم. سپس از متد **DoAction** از کلاس یا استراکت **Orc** متد **Execute** از اینترفیس مد نظر را که شی کانکریت ساخته شده در آن است، **call** کردیم و با تغییر استراتژی توسط مد **SetStrategy** و گرفتن شی جدیدی از کانکریت های **ICombat** توانستیم استراتژی را تغییر دهیم. در این ساختار، تمام تصمیمات و اعمال مورد نیاز شما در **Execute** اتفاق می افتد و هیچ تصمیم شرطی یا ... در **DoActions** نمی افتد! با این استراتژی، هر کلاس یا استراکت جدید از زیر گروه های **ICombat** که ساخته شود، رفتارهای آن در خود آن کلاس پیاده خواهد شد و دیگر به کلاس هایی که قبلا تستشان را به درستی پس داده اند منتشر نمی شوند! در نتیجه تغییر استراتژی های جدید، موجب تغییر تست های قدیمی نمی شوند!

به صورت کلی، الگوی طراحی Strategy به این شکل است و در کلاس Context، اشاره گری به اینترفیس مربوطه است که به شیئی از کلاس کانکریت اشاره می کند:



شباهت ها و تفاوت های الگوهای State و Strategy :

وجه اشتراک (Similarities):

1- ساختار یکسان:

- هر دو از ترکیب (Composition) استفاده می کنند!
- هر دو یک interface/base class برای رفتارهای مختلف دارند!
- هر دو امکان تغییر رفتار در زمان اجرا (Runtime) را فراهم می کنند!

2- پیاده سازی مشابه:

```

// در هر دو الگو
class Context {
    unique_ptr<IStrategyOrState> current;
    void Request() { current->Handle(); }
};
    
```

3- هدف مشترک:

- حذف وابستگی مستقیم (Loose Coupling)
- رعایت اصل Open/Closed
- جداسازی منطق از کلاس اصلی

1- نیت و هدف (Intent):

Strategy

- انتخاب یک الگوریتم از بین چندین الگوریتم
- تمرکز بر چگونگی انجام کار (How)
- مثال: انتخاب روش پرداخت (کارت، نقد، اعتباری)

State

- تغییر رفتار بر اساس وضعیت داخلی
- تمرکز بر چه چیزی در سیستم رخ می‌دهد (What)
- مثال: تغییر رفتار یک سفارش (در انتظار → تأیید → ارسال)

2- کنترل تغییرات:

Strategy

- مشتری (Client) استراتژی را تغییر می‌دهد
- تغییر استراتژی کاملاً آگاهانه و از بیرون

```
orc.SetCombatStrategy(make_unique<MeleeCombat>()); // کاربر تصمیم می‌گیرد
```

State

- خود State ها وضعیت بعدی را تعیین می‌کنند
- تغییر وضعیت معمولاً خودکار و بر اساس منطق داخلی

```
// State درون  
context->SetState(make_unique<ApprovedState>()); // خودش تصمیم می‌گیرد
```

3- ارتباط بین State ها/Strategy ها:

Strategy

- استراتژی‌ها معمولاً مستقل از هم هستند!
- هر استراتژی جزیره‌ای جداگانه است!

State

- State ها به هم وابسته و آگاه از State های دیگرند!
- یک زنجیره یا چرخه از وضعیت ها را تشکیل می دهند! (State ها می دانند بعد از خودشان در چه تابعی چه State ای می آید!)

4- تعداد و نوع تغییرات:

Strategy

- تغییرات کم تعداد و بر اساس نیاز کسب و کار!
- معمولاً یک بار در ابتدای کار یا به ندرت تغییر می کند!

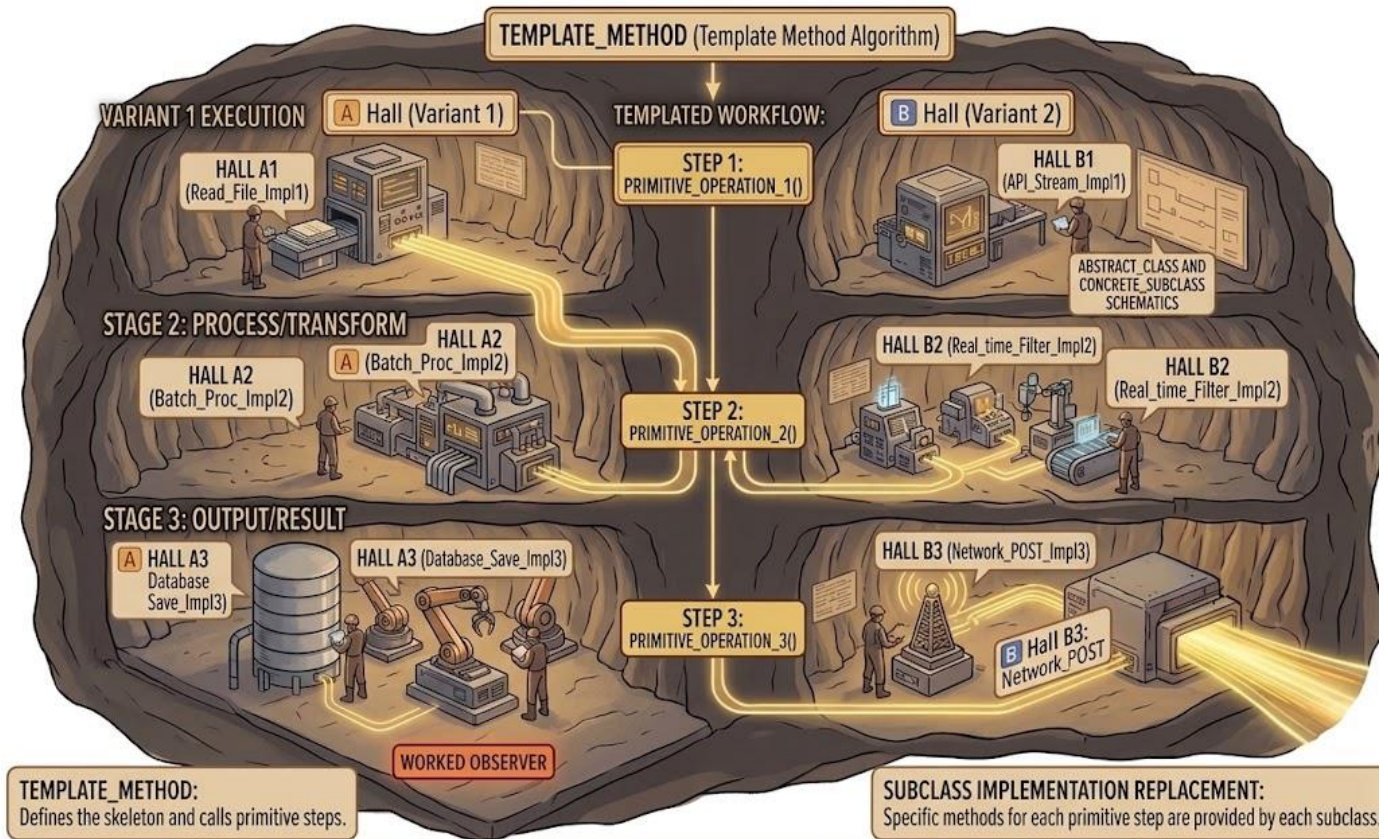
State

- تغییرات پر تعداد و مکرر در طول چرخه حیات
- مدام در حال تغییر بر اساس رویدادها

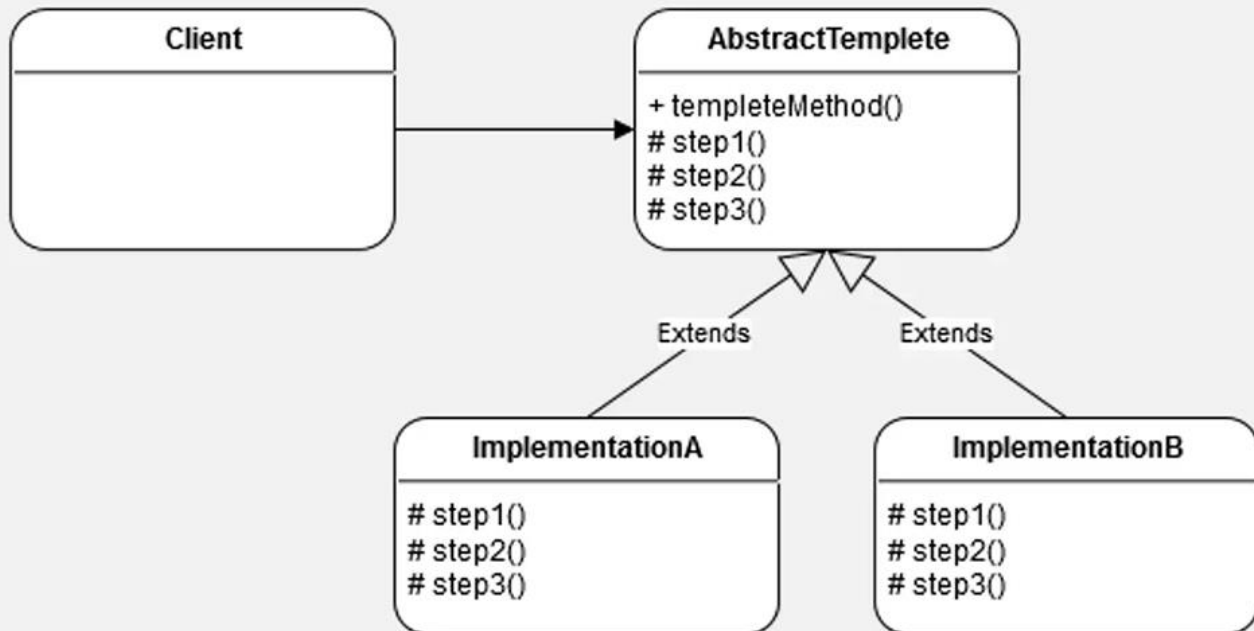
💡 نکته طلایی:

- در الگوی Strategy انتخاب آگاهانه داریم و می گوییم: می خواهم این کار را به این روش انجام دهم!
- در الگوی State واکنش خودکار داریم! یعنی سیستم بر اساس وضعیت فعلی اش این گونه رفتار می کند!

Template Method: پترنی از نوع Behavioral است!



Template Method – Class diagram



چهار چوب یک الگوریتم را تعریف می‌کند و برخی مراحل را به subclassها واگذار می‌کند! Template Method Pattern یکی از الگوهای طراحی رفتاری (Behavioral) است که:

- یک اسکلت (template) از الگوریتم را در یک کلاس پایه تعریف می‌کند.
- مراحل خاص الگوریتم را به متدهای قابل تغییر یا abstract/virtual در کلاس‌های فرزند می‌سپارد.
- به شما اجازه می‌دهد ساختار کلی الگوریتم ثابت بماند اما جزئیات قابل تغییر باشند.

🔧 ساده‌ترش کنیم:

کلاس پدر = دستور پخت نهایی
کلاس‌های فرزند = آشپزهایی که بعضی مراحل را متفاوت اجرا می‌کنند!
به عبارت دیگر: استخراج ساختار ثابت یک الگوریتم و اجازه سفارشی‌سازی مراحل خاص آن توسط کلاس‌های فرزندی.

📌 کاربرد معمول در برنامه‌نویسی: زمانی از این الگو استفاده می‌شود که:

- چندین کلاس الگوریتم یکسانی دارند، اما جزئیات پیاده‌سازی متفاوت است.
- می‌خواهید تکرار کد را کاهش دهید و یکپارچگی را در فرآیند اجرای یک عملیات حفظ کنید. مثلاً در موارد زیر استفاده می‌شود!

- Logging
- Authentication flows
- File processing
- Game AI steps
- Transaction pipelines
- Audit Logging

🎮 سناریوی مثال Audit Log :

فرض کنید سیستم شما برای هر عملیاتی که انجام می‌شود باید یک گزارش حسابرسی (Audit Log) ثبت کند. ساختار کلی ثبت Audit Log همیشه شبیه هم است:

1. گرفتن اطلاعات کاربر
 2. گرفتن اطلاعات عملیات
 3. اعتبارسنجی داده‌ها
 4. ذخیره کردن لاگ
 5. ارسال نوتیفیکیشن (اختیاری)
- اما این مراحل برای عملیات مختلف ممکن است تفاوت داشته باشد.

```
class AuditLogger {
public:
    void LogOperation() {
        auto user = GetUser();
        auto action = GetAction();

        if (!Validate(user, action)) {
            throw std::runtime_error("Invalid audit data");
        }
        Save(user, action);
        // (اختیاری) hook
        Notify(user, action);
    }
protected:
    virtual std::string GetUser() = 0;
    virtual std::string GetAction() = 0;
    virtual bool Validate(const std::string& user, const std::string& action) {
        return true;
    }
    virtual void Save(const std::string& user, const std::string& action) = 0;
    // Hook method - optional override
    virtual void Notify(const std::string& user, const std::string& action) {}
};
```

نکات مهم:

- **LogOperation** الگوریتم نهایی را تعریف می‌کند و معمولاً **virtual** نیست چون مسیر اصلی و رودمپ عملیات را تعریف می‌کند که تغییر ناپذیر است!
- مراحل قابل تغییر به عنوان **virtual / abstract** تعریف می‌شوند!
- **Notify** یک **Hook** است → الزاماً نیازی نیست **override** شود!

✓ قدم دوم – پیاده‌سازی کلاس‌های واقعی

 **مثال 1:** ثبت لاگ برای خرید (Purchase)

```
Class PurchaseAuditLogger {
protected:
    std::string GetUser() override {
        return "User123";
    }
    std::string GetAction() override {
        return "Purchase Item";
    }
    void Save(const std::string& user, const std::string& action) override {
        std::cout << "[Audit] Saving Purchase Log: "
                  << user << " - " << action << "\n";
    }
    void Notify(const std::string&, const std::string&) override {
        std::cout << "Sending purchase notification...\n";
    }
};
```

```
class LoginAuditLogger : public AuditLogger {
protected:
    std::string GetUser() override {
        return "User456";
    }

    std::string GetAction() override {
        return "Login";
    }

    bool Validate(const std::string& user, const std::string& action) override {
        return !user.empty() && action == "Login";
    }

    void Save(const std::string& user, const std::string& action) override {
        std::cout << "[Audit] Saving Login Log: "
            << user << " - " << action << "\n";
    }
};
```

✓ قدم سوم - اجرای الگوریتم

```
int main() {
    PurchaseAuditLogger purchaseLog;
    purchaseLog.LogOperation();

    LoginAuditLogger loginLog;
    loginLog.LogOperation();

    return 0;
}
```

🔑 موارد استفاده:

- هنگامی که چندین کلاس، الگوریتم مشابهی دارند با تفاوت‌های جزئی!
- وقتی نیاز به ثبت لاگ، احراز هویت یا اعتبارسنجی مشترک دارید!
- برای ایجاد فریم ورک‌هایی که رفتار کلی را تعریف می‌کنند!

این الگو به ویژه در سیستم‌های امنیتی، سیستم‌های گزارش‌گیری و فریم ورک‌های پردازش داده بسیار مفید است!

```
class AbstractClass {
public:
    void templateMethod() {                // (در برخی زبان‌ها final روش قالبی)
        step1();                          // مرحله ثابت
        step2();                          // مرحله قابل بازنویسی
        step3();                          // مرحله قابل بازنویسی
    }
protected:
    void step1() { /* پیاده‌سازی ثابت */ }
    virtual void step2() = 0; // اجباری برای override
    virtual void step3() { /* پیش‌فرض قابل override */ }
};
```

فرض کنید یک سیستم مدیریتی دارید که عملیات‌های مختلفی مثل ایجاد کاربر، حذف محصول و به‌روزرسانی تنظیمات را انجام می‌دهد.

شما می‌خواهید:

1- قبل از هر عملیات: یک ورودی در AuditLog با Timestamp و نام عملیات ثبت شود.

2- عملیات اصلی اجرا شود (که بسته به نوع عملیات متفاوت است).

3- بعد از عملیات، نتیجه (موفق/ ناموفق) در لاگ ثبت شود.

اینجاست که Template Method کمک می‌کند!

پیاده‌سازی در C++

```
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>
// کلاس انتزاعی برای تعریف الگوریتم قالبی
class AuditOperation {
public:
    // روش قالبی: ساختار ثابت تمام عملیات‌ها
    void execute(const std::string& operationName) {
        logStart(operationName);
        bool success = performOperation();
        // این روش توسط subclassها پیاده‌سازی می‌شود
        logEnd(operationName, success);
    }
protected:
    // روش‌های خالص مجازی: باید در subclassها پیاده‌سازی شوند
    virtual bool performOperation() = 0;
private:
    // مرحله ثابت: ثبت شروع عملیات
    void logStart(const std::string& opName) {
        std::cout << "[AUDIT] Starting operation: " << opName
                    << " at " << getCurrentTime() << std::endl;
    }
    // مرحله ثابت: ثبت پایان عملیات
    void logEnd(const std::string& opName, bool success) {
        std::cout << "[AUDIT] Operation " << opName
                    << " completed " << (success ? "successfully" : "with failure")
                    << " at " << getCurrentTime() << std::endl;
    }
    std::string getCurrentTime() {
        auto now =
            std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
        std::string timeStr = std::ctime(&now);
        timeStr.pop_back();
        // حذف کاراکتر newline
        return timeStr;
    }
};
```

```
//1: ایجاد کاربر
class CreateUserOperation : public AuditOperation {
protected:
    bool performOperation() override {
        std::cout << "    → Creating user in database..." << std::endl;
        // شبیه سازی عملیات-می تواند true/false برگرداند
        return true; // فرض می کنیم موفق بود
    }
};

//2: حذف محصول
class DeleteProductOperation : public AuditOperation {
protected:
    bool performOperation() override {
        std::cout << "    → Deleting product from inventory..." << std::endl;
        // شبیه سازی: گاهی ناموفق است!
        return false; // فرض می کنیم ناموفق بود
    }
};

//3: به روز رسانی تنظیمات
class UpdateSettingsOperation : public AuditOperation {
protected:
    bool performOperation() override {
        std::cout << "    → Updating system settings..." << std::endl;
        return true;
    }
};

// استفاده در برنامه اصلی
int main() {
    CreateUserOperation createUser;
    DeleteProductOperation deleteProduct;
    UpdateSettingsOperation updateSettings;
    createUser.execute("CreateUser");
    std::cout << "\n";
    deleteProduct.execute("DeleteProduct");
    std::cout << "\n";
    updateSettings.execute("UpdateSettings");

    return 0;
}
```

خروجی نمونه: 

```
[AUDIT] Starting operation: CreateUser at Tue Dec 16 10:30:45 2025
→ Creating user in database...
[AUDIT] Operation CreateUser completed successfully at Tue Dec 16 10:30:45 2025
[AUDIT] Starting operation: DeleteProduct at Tue Dec 16 10:30:45 2025
→ Deleting product from inventory...
[AUDIT] Operation DeleteProduct completed with failure at Tue Dec 16 10:30:45 2025
[AUDIT] Starting operation: UpdateSettings at Tue Dec 16 10:30:45 2025
→ Updating system settings...
[AUDIT] Operation UpdateSettings completed successfully at Tue Dec 16 10:30:45 2025
```

✓ **مزیت اصلی:** اگر بخواهید سیستم AuditLog را تغییر دهید (مثلاً به فایل بنویسید یا به سرور بفرستید)، فقط کافی است کلاس پایه را

ویرایش کنید و هیچ subclassی نیاز به تغییر ندارد!

Hook Methods (روش‌های قلاب): می‌توانید روش‌های اختیاری اضافه کنید تا subclassها بتوانند در صورت نیاز override کنند و در execute() فراخوانی شوند!

```
virtual void onBeforeOperation() {} // پیش فرض خالی  
virtual void onAfterOperation() {} // پیش فرض خالی
```

🔑 استفاده از **Non-virtual Interface (NVI) Idiom**: الگویی در C++ که روش عمومی غیرمجازی دارد و روش‌های مجازی را private می‌کند که امن‌تر است. به بیان بهتر، متد اصلی را به صورت عمومی و none virtual تعریف می‌کنیم و تمام متدهای قابل override را private تعریف می‌کنیم!

🔑 کاربرد در سیستم‌های واقعی:

- سیستم‌های تراکنش بانکی

- پردازش فایل‌های مختلف با ساختار یکسان (open → process → close)

- تست‌های واحد (setup → test → teardown)

📌 جمع‌بندی

- Template Method ساختار الگوریتم را در یک جا نگه می‌دارد!

- از تکرار کد جلوگیری می‌کند!

- تضمین می‌کند که مراحل حیاتی (مثل AuditLog) همیشه اجرا شوند!

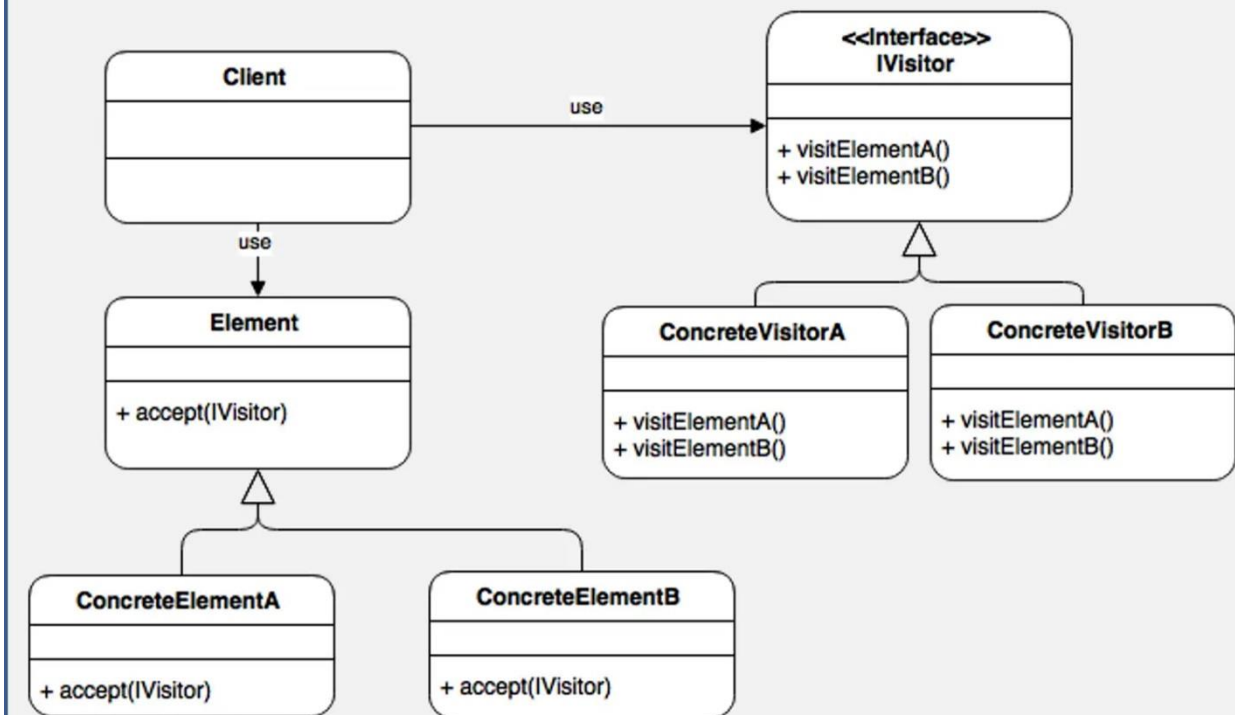
- یکی از ساده‌ترین و در عین حال مؤثرترین الگوهای طراحی برای کنترل جریان اجرایی است!

- امکان سفارشی‌سازی مراحل را ممکن می‌سازد!

- استفاده از Hook Methods برای منطق اختیاری (متدهای هوک متدهای virtual غیر محض هستند که عملیاتی را انجام می‌دهند و اختیار override کردنشان را نیز به subclass هایشان می‌دهند!)



Visitor pattern – Class diagram



این الگو به شما اجازه می‌دهد عملیات جدیدی را روی مجموعه‌ای از اشیاء (معمولاً یک ساختار درختی یا شبکه‌ای) بدون تغییر کلاس‌های آن اشیاء اضافه کنید.

مثالهایی از کاربردهای عملی:

- 1- هایلایت کردن تگ‌ها یا تبدیل ساختار HTML به متن ساده
- 2- در کامپایلرها برای تحلیل و تبدیل کد منبع
- 3- در پردازش اسناد برای تبدیل فرمت‌های مختلف اسناد
- 4- در رندرینگ برای رندر اشیاء به فرمت‌های مختلف مثل رندر کارتونی، لگویی، واقعی و ...
- 5- در تحلیل کد برای جمع‌آوری آمار و متریک‌های کد.

 فرض کنید یک درخت HTML دارید (div, p, span, img, ...) و می‌خواهید عملیات مختلفی را روی آنها انجام دهید:

- Render ○
- Highlight ○
- Export to Plain Text ○
- SEO Analysis ○

اگر این عملیات را داخل خود کلاس‌ها بگذارید، کلاس‌ها خیلی بزرگ می‌شوند و هر تغییر یعنی تغییر همه کلاس‌ها! اینجاست که Visitor به کمک شما می‌آید و منطق را جدا می‌کند.

مفاهیم کلیدی در Visitor:

Element: رابط یا کلاسی که باید بتواند یک Visitor را بپذیرد ('accept' داشته باشد). به بیان بهتر یعنی کلاس پایه تمام نودها HTMLNode یا NodeInterface

ConcreteElement: کلاس‌های واقعی که عملیات روی آنها انجام می‌شود (مثلاً PTag, DivTag). مثل Div, Paragraph, TextNode

Visitor: رابطی که برای هر ConcreteElement یک متد visit() تعریف می‌کند.

ConcreteVisitor: پیاده‌سازی واقعی الگوریتم‌ها (مثلاً HighlightVisitor, PlainTextVisitor).

```
#include <iostream>
#include <string>
#include <memory>
#include <vector>
#include <sstream>
#include <utility>
// Forward declaration
class HtmlVisitor;
// =====
// Base node
// =====
class HtmlNode {
public:
    virtual ~HtmlNode() = default;
    virtual void accept(HtmlVisitor& visitor) = 0;
};
```

```

// Forward declarations for visitor parameters
class HtmlTextNode;
class HtmlDiv;
class HtmlParagraph;
class HtmlSpan;
// =====
// Visitor interface
// =====
// برای سادگی، visit را pure virtual کردیم تا بتوانیم accept ها را inline بنویسیم
class HtmlVisitor {
public:
    virtual ~HtmlVisitor() = default;

    virtual void visit(HtmlTextNode&) = 0;
    virtual void visit(HtmlDiv&) = 0;
    virtual void visit(HtmlParagraph&) = 0;
    virtual void visit(HtmlSpan&) = 0;
};
// =====
// Base element
// =====
class HtmlElement : public HtmlNode {
protected:
    std::string tag_name_;
    std::vector<std::unique_ptr<HtmlNode>> children_;
public:
    explicit HtmlElement(std::string tag_name)
        : tag_name(std::move(tag_name)) {}

    const std::string& tagName() const {
        return tag_name_;
    }
    void addChild(std::unique_ptr<HtmlNode> child) {
        children_.push_back(std::move(child));
    }
    // Visitor ها برای پیمایش فرزندان از این تابع استفاده می‌کنند
    void acceptChildren(HtmlVisitor& visitor) {
        for (auto& child : children_) {
            child->accept(visitor);
        }
    }
};
// =====
// Concrete nodes
// =====
class HtmlTextNode : public HtmlNode {
    std::string content_;
public:
    explicit HtmlTextNode(std::string content)
        : content_(std::move(content)) {}

    const std::string& content() const {
        return content_;
    }
    void accept(HtmlVisitor& visitor) override {
        visitor.visit(*this);
    }
};

```

```

class HtmlDiv : public HTMLElement {
public:
    HtmlDiv() : HTMLElement("div") {}

    void accept(HtmlVisitor& visitor) override {
        visitor.visit(*this);
    }
};

class HtmlParagraph : public HTMLElement {
public:
    HtmlParagraph() : HTMLElement("p") {}

    void accept(HtmlVisitor& visitor) override {
        visitor.visit(*this);
    }
};

class HtmlSpan : public HTMLElement {
public:
    HtmlSpan() : HTMLElement("span") {}

    void accept(HtmlVisitor& visitor) override {
        visitor.visit(*this);
    }
};

// =====
// Concrete visitors
// =====
// A. Plain Text Extractor
class PlainTextVisitor : public HtmlVisitor {
    std::string result_;
public:
    const std::string& getText() const {
        return result_;
    }
    void visit(HtmlTextNode& text) override {
        result_ += text.content();
    }
    void visit(HtmlDiv& div) override {
        div.acceptChildren(*this);
    }
    void visit(HtmlParagraph& p) override {
        if (!result_.empty()) {
            result_ += '\n';
        }
        p.acceptChildren(*this);
    }
    void visit(HtmlSpan& span) override {
        span.acceptChildren(*this);
    }
};

// B. Highlighted output like [div], [p], [span]
class HighlightVisitor : public HtmlVisitor {
    std::string result_;
public:
    const std::string& getHighlighted() const {
        return result_;
    }
};

```

```

void visit(HtmlTextNode& text) override {
    result_ += text.content();
}
void visit(HtmlDiv& div) override {
    result_ += "[div]";
    div.acceptChildren(*this);
    result_ += "[/div]";
}
void visit(HtmlParagraph& p) override {
    result_ += "[p]";
    p.acceptChildren(*this);
    result_ += "[/p]";
}
void visit(HtmlSpan& span) override {
    result_ += "[span]";
    span.acceptChildren(*this);
    result_ += "[/span]";
}
};
// C. Pretty HTML generator
class HtmlGeneratorVisitor : public HtmlVisitor {
    std::ostringstream oss_;
    int indent_ = 0;
    void writeIndent() {
        for (int i = 0; i < indent_; ++i) {
            oss_ << " ";
        }
    }
public:
    std::string getResult() const {
        return oss_.str();
    }
    void visit(HtmlTextNode& text) override {
        writeIndent();
        oss_ << text.content() << '\n';
    }
    void visit(HtmlDiv& div) override {
        writeIndent();
        oss_ << "<div>\n";
        ++indent_;
        div.acceptChildren(*this);
        --indent_;
        writeIndent();
        oss_ << "</div>\n";
    }
    void visit(HtmlParagraph& p) override {
        writeIndent();
        oss_ << "<p>\n";
        ++indent_;
        p.acceptChildren(*this);
        --indent_;
        writeIndent();
        oss_ << "</p>\n";
    }
    void visit(HtmlSpan& span) override {
        writeIndent();
        oss_ << "<span>\n";
        ++indent_;
        span.acceptChildren(*this);
        --indent_;
        writeIndent();
    }
};

```

```

        oss_ << "</span>\n";
    }
};

class StatsVisitor : public HtmlVisitor { // D. Statistics collector
public:
    struct Stats {
        int totalNodes = 0;
        int textNodes = 0;
        int divs = 0;
        int paragraphs = 0;
        int spans = 0;
        int otherElements = 0;
    } stats;
    void visit(HtmlTextNode&) override {
        ++stats.totalNodes;
        ++stats.textNodes;
    }
    void visit(HtmlDiv& div) override {
        ++stats.totalNodes;
        ++stats.divs;
        div.acceptChildren(*this);
    }
    void visit(HtmlParagraph& p) override {
        ++stats.totalNodes;
        ++stats.paragraphs;
        p.acceptChildren(*this);
    }

    void visit(HtmlSpan& span) override {
        ++stats.totalNodes;
        ++stats.spans;
        span.acceptChildren(*this);
    }
    void print() const {
        std::cout << "=== HTML Document Statistics ===\n"
                  << "Total Nodes: " << stats.totalNodes << '\n'
                  << "Text Nodes: " << stats.textNodes << '\n'
                  << "<div>: " << stats.divs << '\n'
                  << "<p>: " << stats.paragraphs << '\n'
                  << "<span>: " << stats.spans << '\n'
                  << "Other Elements: " << stats.otherElements << '\n';
    }
};

// =====
// Sample document
// =====
std::unique_ptr<HtmlNode> createSampleDocument() {
    auto root = std::make_unique<HtmlDiv>();

    auto title = std::make_unique<HtmlParagraph>();
    title->addChild(std::make_unique<HtmlTextNode>("Welcome to Visitor Pattern!"));
    root->addChild(std::move(title));

    auto content = std::make_unique<HtmlDiv>();

    auto para1 = std::make_unique<HtmlParagraph>();
    para1->addChild(std::make_unique<HtmlTextNode>("This is a sample paragraph with "));
    auto emphasis = std::make_unique<HtmlSpan>();
    emphasis->addChild(std::make_unique<HtmlTextNode>("highlighted text"));
    para1->addChild(std::move(emphasis));
}

```

```

para1->addChild(std::make_unique<HtmlTextNode>("."));
content->addChild(std::move(para1));

auto para2 = std::make_unique<HtmlParagraph>();
para2->addChild(std::make_unique<HtmlTextNode>("Another paragraph for
demonstration."));

content->addChild(std::move(para2));

root->addChild(std::move(content));

return root;
}
// =====
// Main
// =====
int main() {
    auto document = createSampleDocument();

    PlainTextVisitor plainText;
    document->accept(plainText);
    std::cout << "📄 Plain Text Output:\n"
                << plainText.getText() << "\n\n";

    HighlightVisitor highlight;
    document->accept(highlight);
    std::cout << "🔍 Highlighted Output:\n"
                << highlight.getHighlighted() << "\n\n";
    HtmlGeneratorVisitor generator;
    document->accept(generator);
    std::cout << "🔗 Generated HTML:\n"
                << generator.getResult() << '\n';

    StatsVisitor stats;
    document->accept(stats);
    stats.print();

    return 0;
}

```

خروجی کد: 

```

📄 Plain Text Output:
Welcome to Visitor Pattern!
This is a sample paragraph with highlighted text.
Another paragraph for demonstration.

🔍 Highlighted Output:
[div][p>Welcome to Visitor Pattern![/p][div][p>This is a sample paragraph with
[span>highlighted text[/span].[/p][p]Another paragraph for
demonstration.[/p][div][div]

🔗 Generated HTML:
<div>
  <p>
    Welcome to Visitor Pattern!
  </p>
</div>
<div>
  <p>
    This is a sample paragraph with

```

```

    <span>
        highlighted text
    </span>
    .
</p>
<p>
    Another paragraph for demonstration.
</p>
</div>
</div>

```

```

=== HTML Document Statistics ===
Total Nodes: 11
Text Nodes: 5
<div>: 2
<p>: 3
<span>: 1
Other Elements: 0

```

🎯 مزایا و معایب پترن Visitor

✅ مزایا:

- 1- اصل **Open/Closed**: می‌توانید عملیات جدیدی اضافه کنید بدون تغییر کلاس‌های موجود
- 2- جدا کردن **مسئولیت‌ها**: منطق عملیات از ساختار داده جدا می‌شود
- 3- جمع‌آوری **اطلاعات**: می‌توان اطلاعات را از اشیاء مختلف جمع‌آوری کرد
- 4- **انعطاف‌پذیری**: می‌توانید Visitor جدیدی مثل ToMarkdownVisitor یا ToXMLVisitor اضافه کنید بدون تغییر در کلاس‌های تگ.
- 5- رعایت **Single Responsibility Principle**: هر Visitor فقط یک کار انجام می‌دهد.
- 6- **قابلیت تست بالا**: هر الگوریتم به صورت جداگانه قابل تست است.

❌ معایب:

- 1- **پیچیدگی**: اضافه کردن کلاس element جدید نیاز به تغییر تمام visitor ها دارد
- 2- **دسترسی محدود**: visitorها فقط به متدهای public عناصر دسترسی دارند
- 3- زمانی که کلاس‌های ConcreteElement زیاد می‌شوند، باید visit را در تمام Visitor ها به‌روز کنید.
- 4- **Circular dependency** در کدهای بزرگ می‌تواند چالش‌برانگیز باشد (استفاده از forward declaration و ساختارهای هوشمند مثل std::variant یا std::any گاهی جایگزین می‌شود).
- 5- Visitor فقط وقتی کاربرد دارد که ساختار شما پایدار باشد (تعداد Element ها زیاد تغییر نکند).
- 6- **افزایش تعداد کلاس‌ها**

📌 جمع‌بندی:

همان‌طور که مشاهده کردید، هر یک از عناصر ساختار داده (Object Structure) با پیاده‌سازی متد accept، شیء Visitor را دریافت می‌کنند و آن را به متد visit مناسب هدایت می‌کنند. از آنجا که کلاس Visitor، متد visit را برای انواع مختلف عناصر Overload کرده

است، هر عنصر متناسب با نوع واقعی خود پردازش می‌شود. به این ترتیب، بدون نیاز به تغییر کلاس‌های موجود، می‌توان رفتارهای جدیدی را تنها با ایجاد Visitor های جدید به سیستم اضافه کرد.

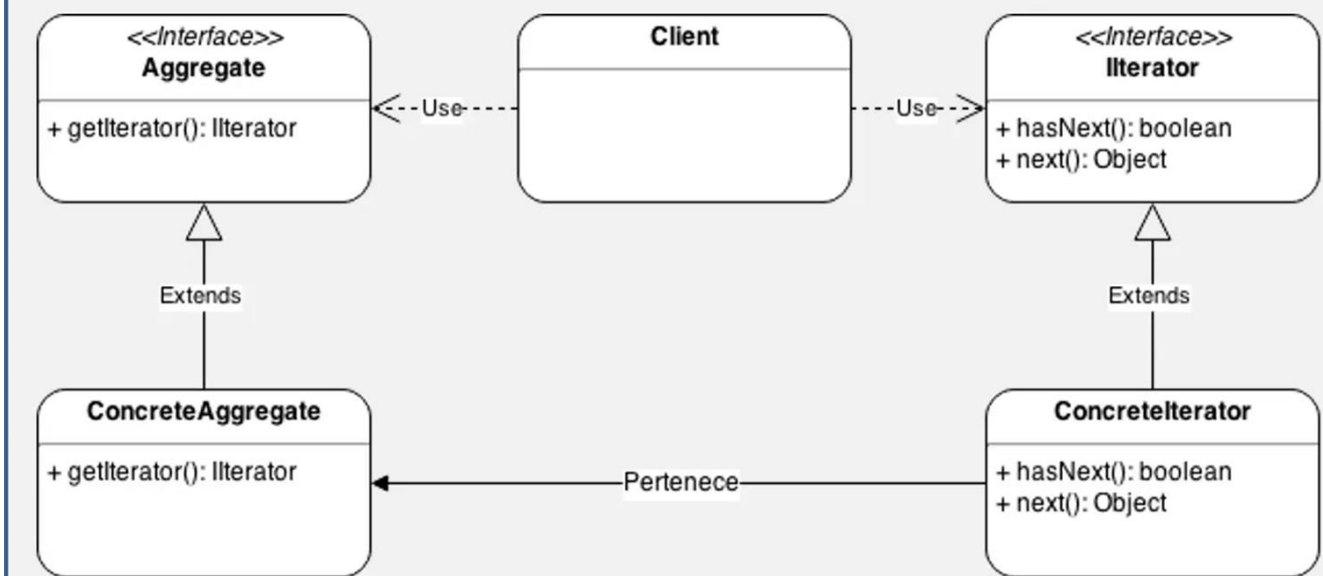
یکی از مهم‌ترین کاربردهای الگوی Visitor در موتورهای رندر، کامپایلرها، ابزارهای تحلیل کد و پردازش ساختارهای درختی است.

برای مثال، یک صحنه سه‌بعدی را در نظر بگیرید. این صحنه تنها یک بار ساخته می‌شود، اما می‌توان آن را با موتورهای رندر مختلف پردازش کرد. یک موتور ممکن است صحنه را به صورت کارتونی (Cartoon Rendering)، دیگری به صورت واقع گرایانه یعنی Photorealistic Rendering و موتور دیگری به شکل Wireframe یا حتی با جلوه‌های هنری نمایش دهد. در این حالت، خود ساختار صحنه هیچ تغییری نمی‌کند؛ تنها Visitor یا همان موتور پردازش‌کننده تغییر می‌کند و خروجی متفاوتی تولید می‌شود.

اگر برای هر نوع رندر مجبور بودیم ساختار صحنه را دوباره طراحی یا بازسازی کنیم، توسعه و نگهداری سیستم بسیار پرهزینه، زمان‌بر و مستعد خطا می‌شد. اما الگوی Visitor این امکان را فراهم می‌کند که با ثابت نگه داشتن ساختار داده و تنها با افزودن Visitor های جدید، رفتارهای متنوع و خروجی‌های متفاوتی ایجاد کنیم. این جداسازی «ساختار داده» از «عملیات قابل انجام روی آن و استخراج خروجی‌های مختلف مناسب» مهم‌ترین مزیت الگوی Visitor محسوب می‌شود.



Iterator pattern – Class diagram



این الگو دسترسی به عناصر یک مجموعه را یکپارچه می‌کند و گشت و گزار در ساختارهای داده را بدون نیاز به شناخت جزئیات داخلی آنها کاملاً ممکن می‌سازد!

در دنیای برنامه‌نویسی، اغلب با مجموعه‌هایی از داده‌ها سروکار داریم مانند لیست‌ها، درخت‌ها، پشته‌ها، صف‌ها و غیره. حال چه بخواهیم تمام عناصر را چاپ کنیم، چه بخواهیم به دنبال یک مورد خاص باشیم، نیاز به عبور (Traversal) از مجموعه داریم.

اما چطور می‌توانیم این کار را بدون شکستن اصول کپسوله‌سازی (Encapsulation) انجام دهیم؟ چطور بدون دانستن اینکه داده‌ها در آرایه ذخیره شده‌اند یا لیست پیوندی، بتوانیم روی آن‌ها حرکت کنیم؟ اینجا است که پترن Iterator به کمک ما می‌آید.

هدف پترن Iterator چیست؟ فراهم کردن یک راه یکنواخت برای عبور از یک مجموعه از اشیاء، بدون نیاز به نمایش ساختار داخلی آن مجموعه! این پترن یکی از پترن‌های رفتاری (Behavioral) است و در C++ به صورت بسیار طبیعی و گسترده‌ای در Standard STL Library Template پیاده‌سازی شده است.

ساختار کلی پترن Iterator : الگوی Iterator از چهار مؤلفه اصلی تشکیل شده:

1. **Iterator** یک رابط انتزاعی که عملیات‌های پیمایش مثل `next()`, `hasNext()`, `current()` را تعریف می‌کند.
2. **ConcreteIterator** پیاده‌سازی واقعی این رابط برای یک ساختار داده خاص.
3. **Aggregate** یا **Iterable** یک رابط برای مجموعه‌ای که باید قابل پیمایش باشد معمولاً فقط یک متد `createIterator()`
4. **ConcreteAggregate** ساختار داده واقعی (مثل لیست، درخت، ...) که یک **Iterator** خاص را برمی‌گرداند.

مزایای پترن **Iterator** 

- کپسوله‌سازی ساختار داخلی: کاربر نمی‌داند داده‌ها چگونه ذخیره شده‌اند.
- چندین پیمایش همزمان: می‌توان چند **Iterator** مستقل روی یک مجموعه داشت.
- یکپارچگی رابط: همه مجموعه‌ها با یک رابط پیمایش می‌شوند.
- پشتیبانی از الگوهای دیگر: مانند `for_each`, `find_if` و سایر الگوریتم‌های STL

Iterator در STL C++ یک نگاه سریع: در C++، **Iterator** ها به صورت کلاس‌ها یا اشاره‌گرها پیاده‌سازی می‌شوند و پنج دسته اصلی دارند:

1. **Input Iterator** خواندنی، یک‌بار عبور
2. **Output Iterator** نوشتنی، یک‌بار عبور
3. **Forward Iterator** پیش‌روی چندباره
4. **Bidirectional Iterator** پیش و عقب
5. **Random Access Iterator** دسترسی مستقیم مثل آرایه

اما در این آموزش، ما یک **Iterator** سفارشی برای یک ساختار داده غیر STL پیاده‌سازی می‌کنیم تا منطق پترن را بفهمیم.

 **مثال کامل: کتابخانه‌ی کتاب‌ها با Iterator سفارشی:** فرض کنید یک سیستم کتابخانه داریم که کتاب‌ها را در یک آرایه داخلی نگه می‌دارد. می‌خواهیم بتوانیم :

- همه کتاب‌ها را پیمایش کنیم.
- بدون دسترسی مستقیم به آرایه، فقط با یک رابط استاندارد.

کلاس مدل داده Book: 

```
#include <iostream>
#include <string>
#include <memory>

class Book {
private:
    std::string title;
    std::string author;
public:
    Book(const std::string& t, const std::string& a) : title(t), author(a) {}
    const std::string& getTitle() const { return title; }
    const std::string& getAuthor() const { return author; }

    void display() const {
        std::cout << "\"" << getTitle() << "\" by " << getAuthor() << std::endl;
    }
};
```

رابط یا Iterator انتزاعی: 

```
// Iterator انتزاعی
class Iterator {
public:
    virtual ~Iterator() = default;
    virtual bool hasNext() const = 0;
    virtual Book& next() = 0;
    virtual const Book& current() const = 0;
};
```

کلاس BookCollection یا مجموعه کتاب‌ها 

```
class BookCollection;
// BookCollection ملموس برای Iterator
class BookIterator : public Iterator {
private:
    const BookCollection& collection;
    size_t currentIndex;

public:
    BookIterator(const BookCollection& coll) : collection(coll), currentIndex(0) {}
    bool hasNext() const override {
        return currentIndex < collection.size();
    }
};
```

```

    Book& next() override {
        if (!hasNext()) {
            throw std::out_of_range("No more books!");
        }
        return const_cast<Book&>(collection.getBook(currentIndex++));
    }

    const Book& current() const override {
        if (currentIndex == 0 || currentIndex > collection.size()) {
            throw std::out_of_range("Invalid iterator position!");
        }
        return collection.getBook(currentIndex - 1);
    }
};
// کلاس اصلی مجموعه
class BookCollection {
private:
    static const size_t MAX_BOOKS = 100;
    Book* books[MAX_BOOKS];
    size_t count = 0;

public:
    ~BookCollection() {
        for (size_t i = 0; i < count; ++i) {
            delete books[i];
        }
    }

    void addBook(const std::string& title, const std::string& author) {
        if (count < MAX_BOOKS) {
            books[count++] = new Book(title, author);
        }
    }

    size_t size() const { return count; }

    const Book& getBook(size_t index) const {
        if (index >= count) throw std::out_of_range("Index out of range");
        return *books[index];
    }
    // این متد، الگوی Iterator را فعال می‌کند
    std::unique_ptr<Iterator> createIterator() const {
        return std::make_unique<BookIterator>(*this);
    }
};

```

 **نکته:** از `const_cast` در `next()` استفاده کردیم چون `collection` در `BookIterator` ثابت است، اما ممکن است کاربر بخواهد روی کتاب تغییر اعمال کند. در پیاده‌سازی‌های واقعی، می‌توان دو نوع `Iterator` قابل تغییر و غیرقابل تغییر داشت.

 استفاده از `Iterator` کلاینت:

```

void printAllBooks(const BookCollection& collection) {
    auto it = collection.createIterator();
    while (it->hasNext()) {
        it->next().display();
    }
}

```

```

void findBookByTitle(const BookCollection& collection, const std::string& title)
{
    auto it = collection.createIterator();
    bool found = false;
    while (it->hasNext()) {
        Book& book = it->next();
        if (book.getTitle() == title) {
            std::cout << "Found: ";
            book.display();
            found = true;
            break;
        }
    }
    if (!found) {
        std::cout << "Book not found: " << title << std::endl;
    }
}

```

تابع اصلی: 

```

int main() {
    BookCollection library;

    library.addBook("1984", "George Orwell");
    library.addBook("To Kill a Mockingbird", "Harper Lee");
    library.addBook("The Great Gatsby", "F. Scott Fitzgerald");
    std::cout << "📖 All books in library:\n";
    printAllBooks(library);

    std::cout << "\n🔍 Searching for '1984':\n";
    findBookByTitle(library, "1984");

    std::cout << "\n🔍 Searching for 'Dune':\n";
    findBookByTitle(library, "Dune");

    return 0;
}

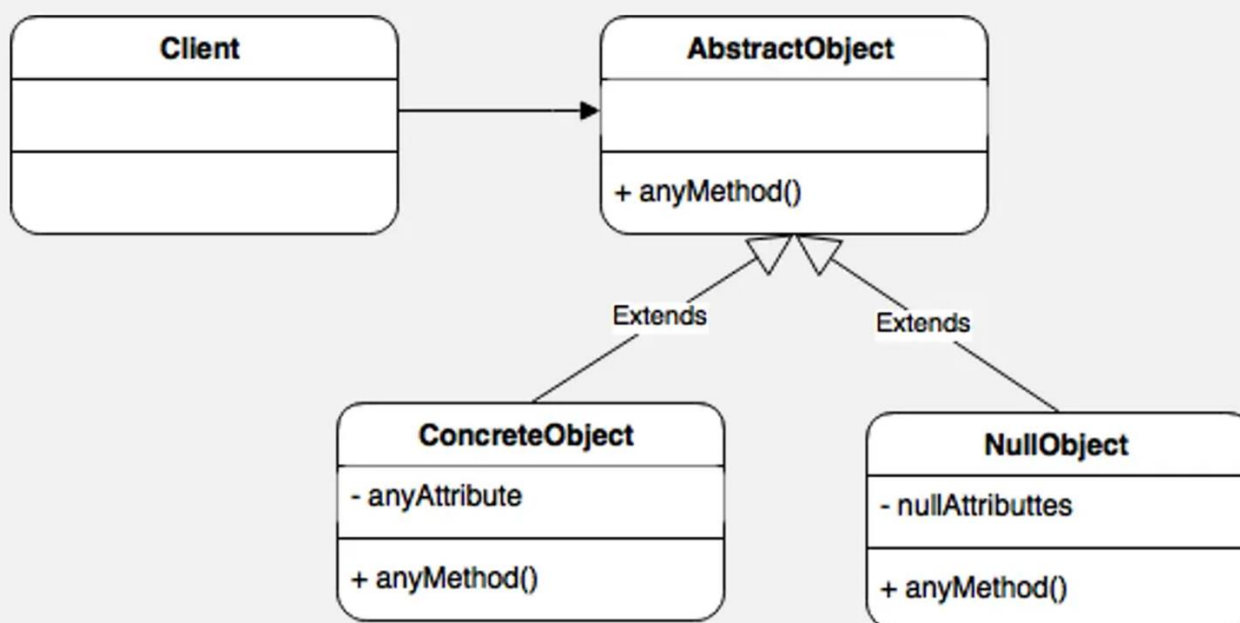
```

 **نتیجه گیری:** همچنان که مشاهده فرمودید، ما می توانیم با داشتن سه متد `GetCurrent()` و `HasNext()` و `Next()` منطق اصلی پیمایش `Item` های یک ساختار `Container` را بدون نقض `Encapsulation` انجام دهیم!

○ الگوی **Null Object** : نوعی الگوی رفتاری است که جزو دسته الگوهای کتاب **Gof** نیست!



Null Object Pattern – Class diagram



در بسیاری از نرم افزارها با موقعیت‌هایی مواجه می‌شویم که ممکن است یک شیء وجود نداشته باشد یا مقدار **null** دریافت شود. راهکار رایج در چنین شرایطی برای جلوگیری از **Runtime Error** ها، استفاده از شرط‌های متعدد برای بررسی مقدار **null** است:

```
if (customer != nullptr)
{
    customer->sendEmail();
}
```

اگر این بررسی‌ها در نقاط مختلف برنامه تکرار شوند، کد به مرور پیچیده، شکننده و سخت‌تر برای نگهداری خواهد شد.

الگوی **Null Object** یک راهکار طراحی است که به جای استفاده از مقدار **null**، یک شیء جایگزین ارائه می‌دهد که رفتار پیش‌فرض و بی‌اثر به عبارتی **(Do Nothing)** دارد. این شیء همان **Interface** یا کلاس پایه شیء اصلی را پیاده‌سازی می‌کند، بنابراین کد مصرف‌کننده بدون نیاز به بررسی **null** می‌تواند با آن کار کند.

 ایده اصلی **Null Object Pattern**: ایده این الگو بسیار ساده است:

به جای اینکه نبود یک شیء را با مقدار **null** نشان دهیم، یک شیء واقعی ایجاد می‌کنیم که همان قرارداد را رعایت می‌کند اما هیچ عملیات خاصی انجام نمی‌دهد. پس هر دو شیء یک **Interface** مشترک دارند، بنابراین استفاده‌کننده تفاوتی بین آن‌ها احساس نمی‌کند.

 مشکل بدون استفاده از **Null Object**: فرض کنید یک سیستم ثبت گزارش (**Logger**) داریم:

```
class Logger {
public:
    virtual void log(const std::string& message) = 0;
};
```

یک **Logger** واقعی:

```
class FileLogger : public Logger {
public:
    void log(const std::string& message) override {
        std::cout << "Writing log: "
                  << message << std::endl;
    }
};
```

حالا در کلاس دیگری از **Logger** استفاده می‌کنیم:

```
class Application {
    Logger* logger;
public:
    Application(Logger* logger)
        : logger(logger) {}

    void run() {
        if(logger != nullptr)
            logger->log("Application started");
    }
};
```

مشکل اینجاست که هر جا از **Logger** استفاده شود، باید بررسی کنیم که آیا مقدار آن **null** است یا خیر. با رشد پروژه، چنین شرطهایی زیاد می‌شوند:

```
if(logger) {  
    logger->log(message);  
}
```

... و این باعث افزایش پیچیدگی کد می‌شود.

🌱 **پیاده‌سازی Null Object:** اکنون یک **Null Logger** ایجاد می‌کنیم:

```
class NullLogger : public Logger {  
public:  
    void log(const std::string& message) override{  
        // هیچ کاری انجام نمی‌شود  
    }  
};
```

حالا **Application** همیشه یک **Logger** معتبر دریافت می‌کند:

```
class Application {  
    Logger* logger;  
public:  
    Application(Logger* logger)  
        : logger(logger) { }  
  
    void run() {  
        logger->log("Application started");  
    }  
};
```

🦷 **استفاده:**

```
FileLogger fileLogger;  
Application app1(&fileLogger);  
app1.run();
```

🦷 **یا:**

```
NullLogger nullLogger;  
Application app2(&nullLogger);  
app2.run();
```

در حالت دوم برنامه بدون خطا اجرا می‌شود، اما عملیات **Logging** انجام نمی‌شود.

Null Object و اصل Polymorphis : قدرت اصلی این الگو از چندریختی (Polymorphism) می‌آید و کد مصرف‌کننده فقط Interface را می‌شناسد:

```
Logger* logger;
```

و اهمیتی نمی‌دهد که شیء واقعی:

- FileLogger است؟
- ConsoleLogger است؟
- NullLogger است؟

این باعث کاهش وابستگی و ساده‌تر شدن کد می‌شود.

 ارتباط با **SOLID** : Null Object ارتباط نزدیکی با چند اصل SOLID دارد.

 اصل **Open/Closed Principle (OCP)** : کلاس‌ها باید برای توسعه باز و برای تغییر بسته باشند.

 بدون **Null Object**:

```
if(logger == nullptr) {  
    رفتار جدید  
}
```

هر بار که نوع جدیدی از Logger اضافه شود، ممکن است مجبور به تغییر کدهای موجود شویم.

 اما با **Null Object** : کد اصلی بدون تغییر باقی می‌ماند.

```
logger->log(message);
```

 اصل **Liskov Substitution Principle (LSP)** : Null Object باید بتواند جایگزین شیء واقعی شود و به نوعی رابطه **is a** بین null object و انواع اشیای نوع لاگر (در مثال ما) وجود دارد! یعنی **Logger*** هم باید بتواند **FileLogger** دریافت کند و هم بتواند **NullLogger** دریافت کند!

 مثال واقعی در **C++** : فرض کنید یک سیستم پرداخت داریم:

```
class PaymentProcessor {  
public:  
    virtual bool pay(double amount)=0;  
};
```

```
class CreditCardPayment : public PaymentProcessor {
public:
    bool pay(double amount) override {
        std::cout<< "Payment: " << amount;
        return true;
    }
};
```

```
class NullPayment : public PaymentProcessor {
public:
    bool pay(double amount) override {
        return true;
    }
};
```

حالا سیستم سفارش همیشه یک PaymentProcessor دارد:

```
class Order {
    PaymentProcessor* payment;
public:
    Order(PaymentProcessor* payment): payment(payment){}

    void checkout()
    {
        payment->pay(100);
    }
};
```

دیگر نیازی نیست بنویسید:

```
if(payment != nullptr)
```

مزایای Null Object Patter: ✔

1- حذف شرط‌های اضافی و کاهش کدهایی نظیر:

```
if(object != nullptr)
```

2- کاهش احتمال خطا: خطاهایی مانند خطای زیر کاهش می‌یابد:

```
NullPointerException! Segmentation Fault!
```

3- افزایش خوانایی کد: کد به جای تمرکز روی بررسی حالت‌های خاص، روی منطق اصلی تمرکز می‌کند.

4- استفاده بهتر از Polymorphism: به جای شرط زیر از رفتار متفاوت اشیاء استفاده می‌شود!

```
if(type == X)
```

❌ معایب Null Object Patter : با وجود مزایا، این الگو همیشه مناسب نیست.

1- ممکن است خطاها را پنهان کند: گاهی نبودن یک شیء باید خطا محسوب شود! مثلاً اگر DatabaseConnection

وجود نداشته باشد، شاید بهتر باشد برنامه متوقف شود، نه اینکه یک Null Object جایگزین شود.

2- ممکن است رفتار غیرمنتظره ایجاد کند! برنامه اجرا می‌شود اما عملیات انجام نمی‌شود و ممکن است تشخیص مشکل سخت شود.

3- تعداد کلاس‌ها افزایش پیدا می‌کند: برای هر Interface ممکن است یک Null Implementation ایجاد شود.

🌱 راه حل اطمینان از عدم امکان Null بودن شیء: حتی اگر کاربر بخواهد به جای یک شیء غیر Null از کلاس مدنظر ما، یک شیء nullptr به constructor یا setter کلاس ارسال کند، بهتر است هر جا در درون یا بیرون کلاس که می‌خواهید به آن شیء دسترسی پیدا کنید، این دسترسی را به واسطه یک getter به آن شیء میسر کنید و در داخل getter مد نظر هم می‌توانید بنویسید!

```
CustomObject* GetCustomObject() const {
    if (!_custom_object) _custom_object = new CustomNullObject();
    // دستور بالا شیء نوع خالی از کاستوم کلاس را می‌سازد، دستور بالا فقط یک بار می‌تواند اجرا شود آن هم به شرطی که شما از بیرون یک نال به این شیء پاس دهید
    return _custom_object;
}
```

🔗 تفاوت Null Object و Optional : در C++ مدرن گاهی به جای Null Object از std::optional استفاده می‌شود. مثلاً:

```
std::optional<Customer>
```

اما اینها دو هدف متفاوت دارند! Std::optional می‌گوید ممکن است مقداری وجود داشته باشد و ممکن است مقداری وجود نداشته باشد! ولی Null Object می‌گوید همیشه مقداری (شیء) وجود دارد هر چند شاید رفتار خنثی داشته باشد!

✅ چه زمانی از Null Object استفاده کنیم؟ وقتی استفاده کنیم که:

- نبودن یک شیء یک حالت معتبر در سیستم است!
- می‌خواهیم بررسی شرط‌های null نبودن را حذف کنیم!
- Interface مشخصی از شیء Null شدنی داریم .
- رفتار پیش‌فرض قابل تعریف است .

🎨 مثال‌ها:

- Logger خاموش
- سیستم Cache غیرفعال
- Notification غیرفعال
- Payment آزمایشی
- Authentication ناشناس

✗ چه زمانی استفاده نکنیم؟ در موارد زیر، از این الگو استفاده نکنید:

- نبود شیء نشان‌دهنده یک خطای جدی است!
- عملیات بدون آن شیء معنی ندارد!
- نیاز دارید کاربر را از مشکل مطلع کنید! (تمامی این سه مورد را در مثال **Auto Pilot** سیستم‌های خودران مثل خودروهای خودران و هواپیمای خودران در نظر بگیرید که باید شیء خلبان هوشمند به کلاس اتاق کنترل وسیله نقلیه وارد و تنظیم شود وگرنه خودرو یا هواپیما حرکت نمی‌کنند یا اگر به هر دلیلی **nullptr** شوند، سیستم دچار وضعیت خطرناک می‌شود! در این شرایط **Null Object** می‌تواند بحران آفرین باشد!)

✗ **جمع‌بندی:** الگوی **Null Object** یک الگوی طراحی ساده اما بسیار کاربردی است که با جایگزین کردن یک شیء خنثی به جای مقدار **null**، نیاز به شرط‌های تکراری را کاهش می‌دهد. این الگو با استفاده از **Polymorphism** باعث کاهش **Coupling**، افزایش خوانایی و سازگاری بیشتر با اصول **SOLID** می‌شود. در **C++**، این الگو در کنار تکنیک‌هایی مانند **RAII**، **Dependency Injection** و **Interface-based Design** می‌تواند به طراحی سیستم‌های قابل نگهداری کمک کند.

فصل چهارم

الگوهای سازمانی (Enterprise Patterns)

الگوهای سازمانی به مجموعه‌ای از راه‌حل‌ها و الگوهای معماری/طراحی گفته می‌شود که برای حل مسائل رایج در سیستم‌های سازمانی (Enterprise Applications) به کار می‌روند؛ سیستم‌هایی که معمولاً پیچیده، بزرگ، چندکاربره و دارای نیازهای جدی مثل تراکنش، امنیت،

یکپارچگی و مقیاس‌پذیری هستند!

یکی از منابع کلاسیک و بسیار معروف در این زمینه کتاب Patterns of Enterprise Application Architecture اثر معمار فقید عرصه نرم افزار، Martin Fowler است.

🌱 دسته‌بندی مهم Enterprise Patterns:

Fowler الگوها را تقریباً در چند گروه اصلی مطرح می‌کند:

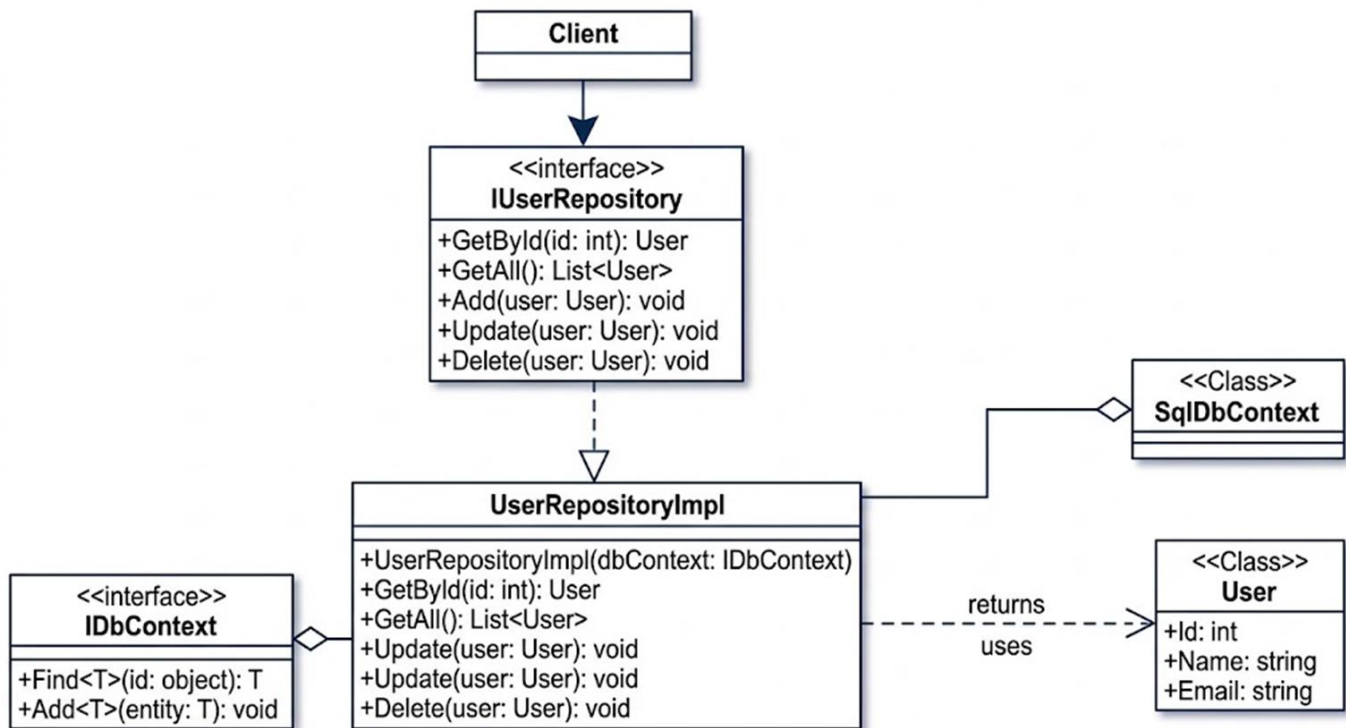
دسته	مثال	کاربرد
Domain Logic	Transaction Script, Domain Model	سازمان‌دهی منطق کسب‌وکار
Data Source	Table Data Gateway, Data Mapper, Active Record	ارتباط با دیتابیس
Presentation	Page Controller, Front Controller, MVC	مدیریت لایه ارائه
Distribution	Data Transfer Object, Remote Facade	ارتباط بین سرویس‌ها/فرآیندها
Concurrency	Pessimistic Offline Lock, Optimistic Offline Lock	کنترل همزمانی
Session State	Server Session State, Client Session State	مدیریت وضعیت کاربر
Base Patterns	Layer Supertype, Mapper, Gateway	ساختاردهی ارتباطات و کد

💡 **نکته:** مرز میان الگوهای معماری (Architectural Patterns) در فصل 5 این کتاب) و الگوهای سازمانی (فصل پیش رو) در منابع مختلف یکسان نیست و برخی الگوها، مانند MVC، بسته به منبع در هر دو حوزه مطرح می‌شوند.

💡 **نکته:** کتابی که پیش روی شماست، با هدف ارائه مرجعی جامع برای تمامی معماری‌ها و الگوهای موجود در دسته‌بندی‌های گوناگون نگاشته نشده است. هدف اصلی این اثر، ارائه تصویری کلی از دسته‌بندی الگوهای نرم‌افزاری و تشریح و بررسی برخی از مهم‌ترین و کاربردی‌ترین نمونه‌های آن‌هاست؛ به گونه‌ای که خواننده بتواند با شناخت بهتر این الگوها، مسیر مناسبی برای بهره‌گیری از آن‌ها در طراحی و توسعه نرم‌افزار بیابد. از همین رو، تلاش کرده‌ایم با معرفی و بررسی مجموعه‌ای منتخب از الگوهای طراحی، معماری و سازمانی، دریچه‌ای به سوی دنیای گسترده الگوهای نرم‌افزاری بگشاییم و زمینه‌ای برای مطالعه و استفاده عمیق‌تر از این مفاهیم فراهم کنیم.

در همین راستا، در این بخش به سراغ سه مورد از شناخته‌شده‌ترین و پرکاربردترین الگوهای معماری سازمانی، یعنی Unit of Repository، Service Locator و Work می‌رویم و ضمن معرفی آن‌ها، به تشریح ساختار، کاربرد و نحوه استفاده از هر یک خواهیم پرداخت و تلاش خواهیم کرد در آینده (به شرط توفیق الهی)، موارد بیشتری همچون Data Mapper، Active Record، Transaction Script، Service Layer، Front Controller، InBox/OutBox Pattern و ... را نیز بررسی کنیم.

پترن Repository: یکی از انواع الگوهای سازمانی یا Enterprise Patterns است!



پترن ریپازیتوری یک الگوی سازمانی نرم افزار است که به عنوان یک لایه واسط بین منطق کسب و کار برنامه و ذخیره سازی داده ها عمل می کند و هدف اصلی آن ارائه یک روش ساختارمند و استاندارد برای دسترسی، مدیریت و دستکاری داده ها است، در حالی که جزئیات زیربنایی فناوری های ذخیره سازی داده را پنهان می کند.

با استفاده از پترن ریپازیتوری، می توانید برای هر جدول دیتابیس یک کلاس ریپازیتوری ایجاد کنید که متدهایی را برای اضافه، حذف، ویرایش و جستجوی داده ها (عملیات CRUD) در آن جدول فراهم می کند. این کلاس ها می توانند از تکنولوژی های مختلفی مانند JPA، JDBC، Hibernate و غیره برای ارتباط با دیتابیس استفاده کنند. این کلاس ها همچنین می توانند از الگوی DAO برای جداسازی منطق دسترسی به داده ها از منطق کسب و کار استفاده کنند.

برای نمونه، فرض کنید شما یک جدول دیتابیس به نام Product دارید که محصولات یک فروشگاه را نگهداری می کند. شما می توانید یک کلاس ریپازیتوری به نام ProductRepository ایجاد کنید! سپس در آن متدهایی مانند addProduct، deleteProduct، updateProduct و findProductById را داشته باشد. این کلاس می تواند از یک کلاس DAO به نام ProductDao استفاده کند که مسئول اجرای کوئری های SQL برای دسترسی به جدول Product است. البته می تواند نام Repository هم داشته باشد که در این مورد، نام Repository مصطلح تر است! که زیر یک نمونه ساده از این کلاس ها را نشان می دهد:

```
#include <string>
#include <vector>
#include <memory>
#include <optional>
#include <iostream>
#include <algorithm>
#include <stdexcept>
#include <string_view>
#include <map>
#include <cassert>

// =====
//   لایه دامنه (Domain Layer)
// =====

class Product {
private:
    long id_;
    std::string name_;
    double price_;
public:
    Product(long id, std::string_view name, double price)
        : id_(id), name_(name), price_(price) {
        if (price < 0.0) throw std::invalid_argument("Price cannot be negative");
    }

    // استفاده از [[nodiscard]] برای جلوگیری از نادیده گرفتن مقادیر بازگشتی
    [[nodiscard]] long getId() const { return id_; }

    // استفاده از string_view برای کارایی بیشتر در C++17
    [[nodiscard]] std::string_view getName() const { return name_; }
    [[nodiscard]] double getPrice() const { return price_; }

    void updatePrice(double newPrice) {
        if (newPrice < 0.0) throw std::invalid_argument("Price cannot be negative");
        price_ = newPrice;
    }
};

// =====
```

```

//اینترفیس (Repository قرارداد لایه دامنه)
// =====
class IProductRepository {
public:
    virtual ~IProductRepository() = default;
    virtual void add(const Product& product) = 0;
    virtual void update(const Product& product) = 0;
    virtual void remove(long id) = 0;
    [[nodiscard]] virtual std::optional<Product> findById(long id) const = 0;
    [[nodiscard]] virtual std::vector<Product> findAll() const = 0;
};
// =====
// پیاده سازی در حافظه (برای تست سریع و Unit Test)
// =====
class InMemoryProductRepository : public IProductRepository {
private:
    std::vector<Product> products_;
public:
    void add(const Product& product) override {
        if (findById(product.getId()).has_value()) {
            throw std::invalid_argument("Product with this ID already exists");
        }
        products_.push_back(product);
    }

    void update(const Product& product) override {
        auto it = std::find_if(products_.begin(), products_.end(),
            [id = product.getId()](const Product& p) { return p.getId() == id; });

        if (it != products_.end()) {
            *it = product; //replace input arg in data real place using iterator
        } else {
            throw std::invalid_argument("Cannot update: Product not found");
        }
    }

    void remove(long id) override {
        products_.erase(
            std::remove_if(products_.begin(), products_.end(),
                [id](const Product& p) { return p.getId() == id; }),
            products_.end()
        );
    }

    [[nodiscard]] std::optional<Product> findById(long id) const override {
        auto it = std::find_if(products_.begin(), products_.end(),
            [id](const Product& p) { return p.getId() == id; });
        return (it != products_.end()) ? std::optional<Product>(*it) : std::nullopt;
    }

    [[nodiscard]] std::vector<Product> findAll() const override {
        return products_;
    }
};

// =====
// پیاده سازی موک برای مطالعه بیشتر است

```



```

// پیاده سازی (MySQL با Mock Connection برای آموزش)
// در پروژه واقعی، این Mock با MySQL Connector/C++ جایگزین می‌شود
// Mock: شبیه ساز ساده رفتار ResultSet و PreparedStatement
// =====
class MockResultSet {
private:
    std::vector<std::tuple<long, std::string, double>> rows_;
    size_t currentIndex_ = 0;
public:
    MockResultSet(std::vector<std::tuple<long, std::string, double>> rows)
        : rows_(std::move(rows)), currentIndex_(0) {}

    bool next() {
        return currentIndex_ < rows_.size() ? ++currentIndex_ > 0 : false;
    }

    long getLong(const std::string&) const {
        return std::get<0>(rows_[currentIndex_ - 1]); }
    std::string getString(const std::string&) const {
        return std::get<1>(rows_[currentIndex_ - 1]); }
    double getDouble(const std::string&) const {
        return std::get<2>(rows_[currentIndex_ - 1]); }
};

class MockPreparedStatement {
private:
    // شبیه سازی حافظه دیتابیس به صورت shared
    std::shared_ptr<std::map<long, std::tuple<long, std::string, double>>> db_;
    std::string operation_; // "INSERT", "UPDATE", "DELETE", "SELECT"
    long queryId_ = 0;
    std::string queryName_;
    double queryPrice_ = 0.0;
public:
    MockPreparedStatement(
        std::shared_ptr<std::map<long, std::tuple<long, std::string, double>>> db,
        const std::string& sql) : db_(db) {
        if (sql.find("INSERT") != std::string::npos) operation_ = "INSERT";
        else if (sql.find("UPDATE") != std::string::npos) operation_ = "UPDATE";
        else if (sql.find("DELETE") != std::string::npos) operation_ = "DELETE";
        else operation_ = "SELECT";
    }

    void setLong(int, long val) {
        if (operation_ == "INSERT") queryId_ = val;
        else if (operation_ == "SELECT" || operation_ == "DELETE" ||
            operation_ == "UPDATE")
            queryId_ = val;
    }

    void setString(int, const std::string& val) { queryName_ = val; }
    void setDouble(int, double val) { queryPrice_ = val; }
};

```

```

void executeUpdate() {
    if (operation_ == "INSERT") {
        (*db_)[queryId_] = {queryId_, queryName_, queryPrice_};
    } else if (operation_ == "UPDATE") {
        if (db_>count(queryId_)) {
            (*db_)[queryId_] = {queryId_, queryName_, queryPrice_};
        } else {
            throw std::runtime_error("Cannot update: Product not found");
        }
    } else if (operation_ == "DELETE") {
        db_>erase(queryId_);
    }
}

std::unique_ptr<MockResultSet> executeQuery() {
    std::vector<std::tuple<long, std::string, double>> results;
    if (operation_ == "SELECT") {
        if (queryId_ > 0) {
            // SELECT WHERE id = ?
            if (db_>count(queryId_)) {
                results.push_back((*db_)[queryId_]);
            }
        } else {
            // SELECT *
            for (const auto& pair : *db_) {
                results.push_back(pair.second);
            }
        }
    }
    return std::make_unique<MockResultSet>(results);
}

};
//MySQL Connection --- شبيه ساز --- :Mock
class MockMySQLConnection {
private:
    // حافظه مشترک برای شبيه سازى ديتابيس واقعى
    std::shared_ptr<std::map<long, std::tuple<long, std::string, double>>> db_;
public:
    MockMySQLConnection()
        : db_(std::make_shared<std::map<long, std::tuple<long, std::string, double>>>()) {}

    std::unique_ptr<MockPreparedStatement> prepareStatement(const std::string& sql)
    {
        return std::make_unique<MockPreparedStatement>(db_, sql);
    }
};

//MySQL Repository --- کلاس اصلى ريبازيتورى مای اسکیوال ---
class MySQLProductRepository : public IProductRepository {
private:
    std::shared_ptr<MockMySQLConnection> connection_;
public:
    explicit MySQLProductRepository(std::shared_ptr<MockMySQLConnection> conn)
        : connection_(std::move(conn)) {}

    void add(const Product& product) override {
        std::string sql = "INSERT INTO products (id, name, price) VALUES (?, ?, ?)";
        auto stmt = connection_>prepareStatement(sql);
        stmt->setLong(1, product.getId());
        stmt->setString(2, std::string(product.getName()));
        stmt->setDouble(3, product.getPrice());
        stmt->executeUpdate();
    }
}

```

```

void update(const Product& product) override {
    std::string sql = "UPDATE products SET name = ?, price = ? WHERE id = ?";
    auto stmt = connection_>prepareStatement(sql);
    stmt->setString(1, std::string(product.getName()));
    stmt->setDouble(2, product.getPrice());
    stmt->setLong(3, product.getId());
    stmt->executeUpdate();
}

void remove(long id) override {
    std::string sql = "DELETE FROM products WHERE id = ?";
    auto stmt = connection_>prepareStatement(sql);
    stmt->setLong(1, id);
    stmt->executeUpdate();
}

[[nodiscard]] std::optional<Product> findById(long id) const override {
    std::string sql = "SELECT id, name, price FROM products WHERE id = ?";
    auto stmt = connection_>prepareStatement(sql);
    stmt->setLong(1, id);
    auto resultSet = stmt->executeQuery();

    if (resultSet->next()) {
        return Product(
            resultSet->getLong("id"),
            resultSet->getString("name"),
            resultSet->getDouble("price")
        );
    }
    return std::nullopt;
}

[[nodiscard]] std::vector<Product> findAll() const override {
    std::string sql = "SELECT id, name, price FROM products";
    auto stmt = connection_>prepareStatement(sql);
    auto resultSet = stmt->executeQuery();

    std::vector<Product> products;
    while (resultSet->next()) {
        products.emplace_back(
            resultSet->getLong("id"),
            resultSet->getString("name"),
            resultSet->getDouble("price")
        );
    }
    return products;
}

};
// =====
// لایه سرویس - (Application Service) کاملاً مستقل از نوع دیتابیس
// =====
class ProductService {
private:
    IProductRepository& repository_;
public:
    explicit ProductService(IProductRepository& repo) : repository_(repo) {}

    void applyDiscount(long productId, double discountPercentage) {
        auto productOpt = repository_.findById(productId);
        if (!productOpt.has_value()) {
            std::cout << "[Service] Product with ID " << productId << " not
found.\n";
            return;
        }
    }
}

```

```

        Product product = *productOpt;
        double newPrice = product.getPrice() * (1.0 - discountPercentage / 100.0);
        product.updatePrice(newPrice);
        // فراخوانی متد update روی ریپازیتوری
        repository_.update(product);
        std::cout << "[Service] Discount applied. New price for '"
                    << product.getName() << "' is $" << product.getPrice() << "\n\n";
    }
};
// =====
// تابع اصلی - نمایش قدرت الگوی Repository در تعویض پیاده‌سازی
// =====
int main() {
    std::cout << "=== Scenario 1: Using In-Memory Repository ===\n";
    InMemoryProductRepository inMemoryRepo;
    ProductService serviceInMemory(inMemoryRepo);
    inMemoryRepo.add(Product(1, "Laptop", 1500.0));
    serviceInMemory.applyDiscount(1, 10.0);

    std::cout << "\n=== Scenario 2: Using MySQL Repository (Mocked) ===\n";

    // ساخت connection مشترک و تزریق به Repository
    auto connection = std::make_shared<MockMySQLConnection>();
    MySQLProductRepository mysqlRepo(connection);
    ProductService serviceMySQL(mysqlRepo);

    mysqlRepo.add(Product(2, "Tablet", 1200.0));
    mysqlRepo.add(Product(3, "Smartphone", 800.0));

    // همان سرویس، همان منطق تجاری، اما حالا با دیتابیس MySQL کار می‌کند!!
    std::cout << "[Service] Applying discount via MySQL Repository...\n";
    serviceMySQL.applyDiscount(2, 15.0);

    std::cout << "\n[Service] Fetching all products from MySQL...\n";
    auto allProducts = mysqlRepo.findAll();
    std::cout << "[Service] Found " << allProducts.size() << " products:\n";
    for (const auto& p : allProducts) {
        std::cout << "    - " << p.getName() << ": $" << p.getPrice() << "\n";
    }

    return 0;
}

```

یکی از اصطلاحات معروف در زمینه‌های مرتبط با **Repository**، اصطلاح **DAO** است که می‌تواند مخفف چندین کلمه باشد، اما در زمینه نرم افزار، معمولاً به معنای **Data Access Object** یا شیء دسترسی به داده است. این یک الگوی طراحی است که منطق دسترسی به داده‌ها را از منطق کسب و کار جدا می‌کند و از یک واسطه یا یک کلاس انتزاعی برای ارتباط با منبع داده استفاده می‌کند. این الگوی طراحی به کاهش تکرار کد، افزایش انعطاف پذیری و تسهیل تست و نگهداری کمک می‌کند.

DAO همچنین می‌تواند به معنای **Decentralized Autonomous Organization** یا سازمان خودمختار غیرمتمرکز باشد. این یک سیستم است که برای توزیع تصمیم‌گیری، مدیریت و مالکیت یک موجودیت ایجاد شده است. این قوانین از طریق قراردادهای هوشمند، یعنی الگوریتم‌هایی که زمانی که شرایط خاصی برآورده شوند، اجرا می‌شوند، به کد سازمان نوشته شده‌اند.

✅ **مهم ترین مزیت الگوی Repository:** مهمترین زیبایی و مزیت الگوی ریپازیتوری آنست که می‌توان برای یک **Table** و یا یک **Aggregate** در دیتابیس، چند **Repository** موازی نوشت! مثلاً یکی برای دیتابیس **MySQL** و یکی برای **SQL Server** و دیگری برای **PostgreSQL** و دیگری ... و ... و اگر دیتابیس نرم افزار را از یک نوع به نوع دیگر **Shift** دهید، به سادگی با ریپازیتوری مخصوص همان نوع دیتابیس، تمام عملیات **CRUD** (**Create-Read-Update-Delete**) را بر اساس دیتابیس نوع

انتخابی سفارشی می کنیم و به جای درگیر شدن دوباره با پیچیدگی های کوئری های دیتابیس به صورت **Hard Code**، به سادگی از بیرون با **Call** کردن متدهای مناسب، نیازهایتان را حل کنید! از طرفی بازگشت به دیتابیس که قبلا با آن کار می کردید هم فقط تعویض شی ریپازیتوری است!

 **مهمترین معماری های نرم افزاری که از Repository استفاده می کنند:**

4- معماری **Clean**

5- معماری **Onion**

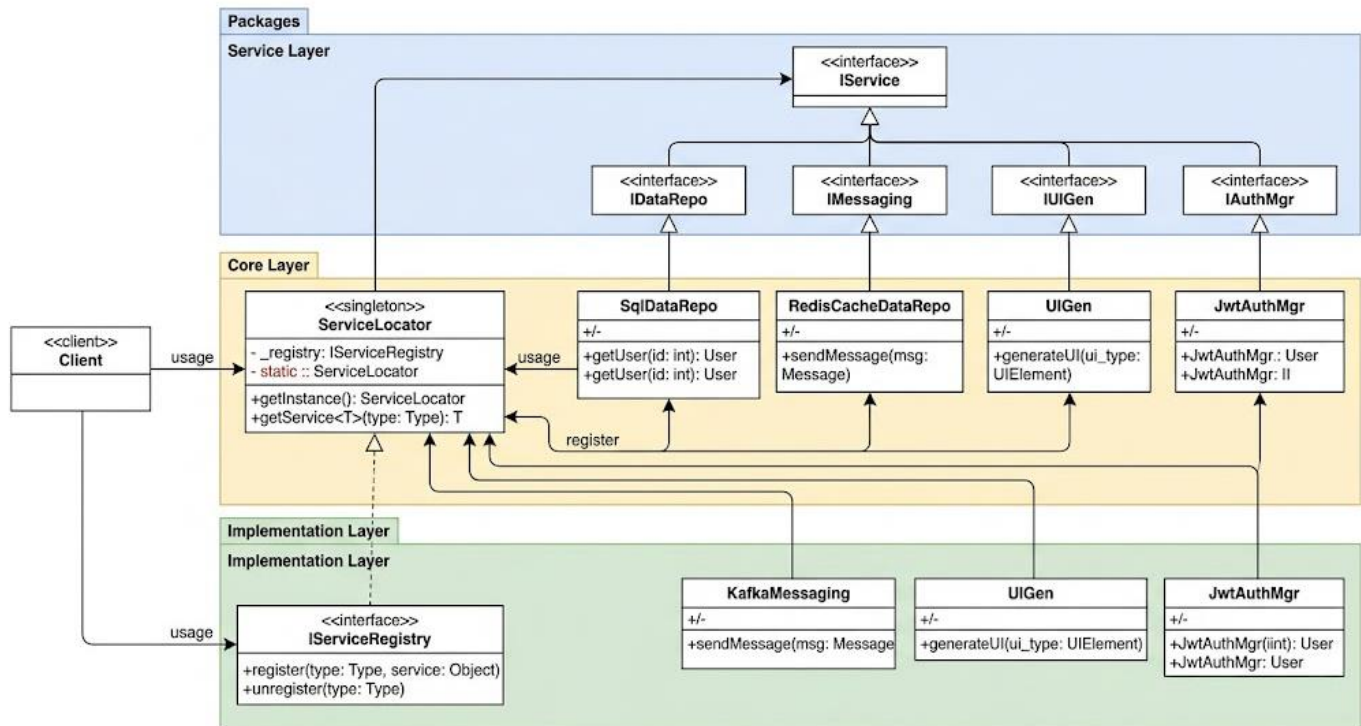
که صد البته هر دو یک معماری هستند که فقط نام لایه هایشان با هم فرق دارد و با منطقی شبیه به منطق فوق (استفاده از **Domain** و استفاده از **Repository** که در دل **Service** قرار گرفته کار عملیات **CRUD** را انجام می دهد!) همچنان که مشاهده می کنید، با تغییر نوع **Repository** می توان دیتابیس مورد استفاده نرم افزار را به سادگی جا به جا کرد! برای تکمیل توضیح این بخش و آگاهی بیشتر در مورد معماری های مذکور، آنجایی که نیاز داریم با دو **Table** دیتابیس تعامل کنیم، اینکه **Repository** را درگیر دو **Table** کنیم صحیح نمی باشد! بلکه باید هر **Repository** فقط با یک **Table** درگیر باشد و در عوض یک **Service** می تواند یک یا دو یا ... **Repository** را دریافت کرده و با آنها تعامل داشته باشد! لذا تعامل با چند **Table** یعنی تعامل یک **Service** با چند **Repository** که هر **Repository** نماینده یک **Table** بوده است!

 **جمع بندی:**

نتیجه اینکه کاربر برنامه نویس در اصل با **Service** ای کار اصلی را انجام می دهد که این **Service** یک شی از انواع **Repository** های مختلف را دریافت می کند! شما سرویس های مورد نیازتان را صدا می زنید و **Service** هم برای تامین نیازتان، متد مورد نیازش از **Repository** را صدا می زند و این **Repository** است که بر اساس نوعش، با دیتابیس مدنظرش ارتباط می گیرد و سپس فرامین **Service** را روی آن دیتابیس اجرا می کند و اگر نتیجه ای را باید **return** کند، آنرا تامین و **return** می کند!

○ پترن **Service Locator**: یکی از الگوهای سازمانی یا Enterprise در نرم افزار است که هدف آن حل یک مسئله ساده اما اساسی است:

چگونه یک شیء بدون دانستن جزئیات پیاده سازی، به وابستگی‌هایی که نیاز دارد دسترسی پیدا کند؟



این الگو در کتاب معروف Core J2EE Patterns (توسط تیم Sun Microsystems) به طور رسمی مستند شد و سال‌ها در معماری‌های سازمانی جاوا (J2EE/Java EE) به‌طور گسترده استفاده می‌شد.

اما نکته جالب درباره این الگو این است که کمتر الگویی در دنیای طراحی نرم‌افزار به اندازه Service Locator محل بحث و مناظره جدی بوده است، به‌طوری که مارتین فاولر (Martin Fowler) و مارک سیمن (Mark Seemann) دیدگاه‌های کاملاً متضادی درباره ارزش آن مطرح کرده‌اند و بسیاری از بزرگان نرم‌افزار، این الگو را نوعی Anti Pattern می‌دانند که در این بخش به دلایل این موضوع خواهیم پرداخت! در این کتاب، ابتدا الگو را کامل توضیح می‌دهیم، سپس مقایسه دقیق آن را با Dependency Injection انجام می‌دهیم، و در نهایت این مناظره تاریخی و دلایل هر دو طرف را بررسی می‌کنیم.

 **بخش اول:** مسئله‌ای که Service Locator حل می‌کند! فرض کنید کلاسی دارید که برای انجام کارش به چند سرویس دیگر نیاز دارد:

```
class Order {
public:
    double total;
    std::string customerEmail;
};

class FileLogger {
public:
    void Log(const std::string& message) {
        // Write log to a file
    }
};

class SmtplibEmailService {
public:
    void SendConfirmation(const std::string& email) {
        // Send confirmation email
    }
};

class StripeGateway {
public:
    void Charge(double amount) {
        // Charge the customer's credit card
    }
};

class OrderProcessor {
public:
    void Process(const Order& order) {
        FileLogger logger;           // Hard-coded dependency
        SmtplibEmailService emailService; // Hard-coded dependency
        StripeGateway paymentGateway; // Hard-coded dependency

        logger.Log("Processing order...");

        paymentGateway.Charge(order.total);

        emailService.SendConfirmation(order.customerEmail);
    }
};
```

✗ این کد چند مشکل جدی دارد:

1- وابستگی محکم **(Tight Coupling): OrderProcessor** مستقیماً به پیاده‌سازی‌های عینی `FileLogger`, `SmtEmailService`, `StripeGateway` وابسته است، نه به انتزاعات.

2- عدم تست‌پذیری: نمی‌توان `OrderProcessor` را بدون ارسال ایمیل واقعی یا اتصال واقعی به `Stripe` تست کرد!

3- عدم انعطاف: تغییر `SmtEmailService` به `SendGridEmailService` نیازمند تغییر کد `OrderProcessor` است.

💡 راه حل با **Service Locator**: `Service Locator` یک رجیستری مرکزی (معمولاً یک `Singleton`) است که مسئول ساخت یا بازگرداندن نمونه‌های سرویس‌های موردنیاز است. به بیان بهتر `Service Locator` مانند کوله پشتی تجهیزات مورد نیاز مورد استفاده تمام بخش‌های یک پروژه نرم افزاری بصورت یکتا است که هر بخش به واسطه آن می‌تواند به سرویس‌های همکار خود متصل شود و به آن دست یابد! پس به جای ساخت مستقیم وابستگی، کلاس از `Locator` درخواست می‌کند:

```
class OrderProcessor {
public:
    void Process(const Order& order) {
        auto& logger = ServiceLocator::Resolve<ILogger>();
        auto& emailService = ServiceLocator::Resolve<IEmailService>();
        auto& paymentGateway = ServiceLocator::Resolve<IPaymentGateway>();

        logger.Log("Processing order...");
        paymentGateway.Charge(order.total);
        emailService.SendConfirmation(order.customerEmail);
    }
};
```

اکنون `OrderProcessor` فقط به `Interface`ها (`ILogger`, `IEmailService`, `IPaymentGateway`) وابسته است، نه پیاده‌سازی خاص که این خودش گامی مهم به سمت اصل وارونگی وابستگی (`DIP`) است.

✪ پیاده‌سازی ساده الگو

```
#include <unordered_map>
#include <typeindex>
#include <any>
#include <stdexcept>

class ServiceLocator {
Private:
    inline static std::unordered_map<std::type_index, std::any> services_;

public:
    template<typename TService>
    static void Register(TService implementation) {
        services_[typeid(TService)] = std::move(implementation);
    }

    template<typename TService>
    static TService Resolve() {
        auto it = services_.find(typeid(TService));
```



```

        if (it != services_.end()){
            return std::any_cast<TService>(it->second);
        }

        throw std::runtime_error("Requested service is not registered.");
    }
};

```

🚀 نمونه پیکربندی در نقطه شروع برنامه (Composition Root):

```

#include <memory>
int main() {
    // Register concrete implementations (performed once during application startup)
    ServiceLocator::Register<std::shared_ptr<ILogger>>(
        std::make_shared<FileLogger>());

    ServiceLocator::Register<std::shared_ptr<IEmailService>>(
        std::make_shared<SmtplibEmailService>());

    ServiceLocator::Register<std::shared_ptr<IPaymentGateway>>(
        std::make_shared<StripeGateway>());

    OrderProcessor processor;

    Order order;
    processor.Process(order);

    return 0;
}

```

🚀 نسخه پیشرفته‌تر با پشتیبانی از Lazy Initialization و Factory و البته Thread Safe:

```

#include <mutex>
#include <memory>
#include <stdexcept>
#include <typeindex>
#include <functional>
#include <shared_mutex>
#include <unordered_map>

class ServiceLocator {
private:
    enum class Lifetime { Singleton, Transient }; // یک شی همه یا اشیای جداگانه قابل
    تکثیر
    struct Registration { // هر نوع شی با هر نوع لایف تایم یک رجیستریشن خواهد داشت
        std::function<std::shared_ptr<void>()> factory;
        // تابع کارخانه که ساخت هر نوع شی را ممکن می‌سازد
        Lifetime lifetime; // آیا شی از نوع سینگلتون است یا قابل تکثیر است
        // Lazy Initialization thread-safe برای
        std::shared_ptr<std::once_flag> once_flag;
        // قفل تردی یک بار مصرف، برای 1 بار ساختن شی کش اینستنس برای این شی
        std::shared_ptr<void> cached_instance; // شی ساخته و کش شده
    };

    std::unordered_map<std::type_index, Registration> registrations_;
    // هش تیپ‌لی که تایپ آی دی را به شی رجیستریشن معادل وصل می‌کند
    mutable std::shared_mutex mutex_; // قفل تردی میوتکس شیرد
    // قفل تردی شیرد نوشتن فقط یک ترد و خواندن همزمان چند ترد را ممکن می‌سازد
    // به بیان بهتر وقتی کسی می‌نویسد، همه تردها حتی خواننده‌ها هم قفل می‌شوند ولی در
    // زمان خواندن، بلاک نمی‌کنند و همه تردها همزمان در زمان غیر نوشتن می‌توانند بخوانند
    // در صورتیکه قبلاً با داشتن میوتکس معمولی و یونیک لاک، چه در زمان خواندن بخش بحرانی
    // مشترک و چه در زمان نوشتن و تغییر در بخش بحرانی مشترک، بلاک و در صف ماندن تمام تردهای

```

فعال در این بخش قطعی بود ولی اینک با شیرد میوتکس، در هنگام نوشتن، همه تردهای خواننده و نویسنده بلاک می شوند ولی در هنگام خواندن بدون تغییر در بخش بحرانی مشترک، عمل بلاک و صف شدن اتفاق نمی افتد

```
public:
// -----
// Factory (Lazy Initialization) با Register Singleton
// این متد یک تابع فکتوری می گیرد و برای اشیای تنبل در زمان نیاز، شی می سازد
// -----
template<typename TService> // (جهت ساخت تنبل در آینده)
void RegisterSingleton(std::function<std::shared_ptr<TService>()> factory) {
    std::unique_lock lock(mutex_);
    // قفل یونیک برای نوشتن، ترد های خواننده و نویسنده در این زمان قفل خواهند شد
    auto type = std::type_index(typeid(TService));
    auto& reg = registrations_[type]; // اگر کلید و ولیو نباشد هم یکی ساخته می شود
    reg.factory = [f = std::move(factory)]() -> std::shared_ptr<void> {
        return std::static_pointer_cast<void>(f());
    };
    reg.lifetime = Lifetime::Singleton;
    reg.once_flag = std::make_shared<std::once_flag>();
    reg.cached_instance = nullptr; // هنوز ساخته نشده، چون تنبل است
}
// -----
// (Eager) Instance با Register Singleton
// این متد برای رجیستر کردن اشیای قبلا ساخته شده به کار می رود
// -----
template<typename TService>
void RegisterSingleton(std::shared_ptr<TService> instance) {
    if (!instance) {
        throw std::invalid_argument("Singleton instance cannot be null.");
    }
    std::unique_lock lock(mutex_);
    // قفل یونیک برای نوشتن، ترد های خواننده و نویسنده در این زمان قفل خواهند شد
    auto type = std::type_index(typeid(TService));

    auto& reg = registrations_[type];
    reg.lifetime = Lifetime::Singleton;
    reg.cached_instance = std::static_pointer_cast<void>(instance);
    reg.once_flag = std::make_shared<std::once_flag>();
    // علامت گذاری به عنوان اجرا شده، اگر فکتوری برای این شی کال شود، دیگر اجرا نمی شود
    std::call_once(*reg.once_flag, []{});
}
// -----
// Register Transient هر بار یک instance جدید می سازد
// -----
template<typename TService>
void RegisterTransient(std::function<std::shared_ptr<TService>()> factory) {
    std::unique_lock lock(mutex_);
    // قفل یونیک برای نوشتن، ترد های خواننده و نویسنده در این زمان قفل خواهند شد
    auto type = std::type_index(typeid(TService));

    auto& reg = registrations_[type];
    reg.factory = [f = std::move(factory)]() -> std::shared_ptr<void> {
        return std::static_pointer_cast<void>(f());
    };
    reg.lifetime = Lifetime::Transient;
    reg.once_flag = nullptr;
    reg.cached_instance = nullptr;
}
```

```

// -----
// Resolve (Thread-Safe & Deadlock-Free)
// متد ریزالو کردن و در اصل درخواست دادن برای دریافت شی مدنظرتان
// -----
template<typename TService>
std::shared_ptr<TService> Resolve() {
    std::shared_lock lock(mutex_); // قفل خواندن که فقط در زمان اجرای سه متد نوشتنی
    // بالا، ترد ها را در این مکان قفل می کند و در غیر از زمان نوشتن، باز می ماند!
    auto type = std::type_index(typeid(TService));
    auto it = registrations_.find(type);
    if (it == registrations_.end()) {
        throw std::runtime_error("Service not registered.");
    }
    // کپی اطلاعات مورد نیاز قبل از آزاد کردن قفل
    auto& reg = it->second;
    auto lifetime = reg.lifetime;
    auto once_flag = reg.once_flag;
    auto factory = reg.factory;
    lock.unlock(); // (جلوگیری از Deadlock) آزاد کردن قفل قبل از ساخت
    // -----
    // std::call_once با Singleton :Lazy Initialization
    // -----
    if (lifetime == Lifetime::Singleton) {
        std::call_once(*once_flag, [&reg]() {
            // این تابع در محیط چند نخه فقط یک بار برای شی وانس فلگ هر شی رجیستریشن اجرا می شود و
            // تضمین ساخته شدن کش اینستنس فقط برای 1 بار است برای اشیایی که
            // نخواهد شد چون در تابع رجیستر برای آنها یک بار با لامبدای خالی اجرا شده است
            reg.cached_instance = reg.factory(); // ایجاد شی کشد
            if (!reg.cached_instance) {
                throw std::runtime_error("Singleton factory returned nullptr.");
            }
        });
        return std::static_pointer_cast<TService>(reg.cached_instance);
    }
    // -----
    // Transient: هر بار یک instance جدید، تابع برای کلاسهای کثیرالانتشار وارد اینجا می شود
    // -----
    auto instance = factory();
    if (!instance) {
        throw std::runtime_error("Transient factory returned nullptr.");
    }
    return std::static_pointer_cast<TService>(instance);
}
// -----
// پاکسازی برای Unit Test
// -----
void Clear() {
    std::unique_lock lock(mutex_);
    registrations_.clear();
}

};

struct ILogger {
    virtual void log(const std::string& msg) = 0;
};

struct ConsoleLogger : ILogger {
    void log(const std::string& msg) override {
        std::cout << msg << std::endl;
    }
};
};

```

```
int main() {
    ServiceLocator locator;
    // Lazy Singleton - فقط وقتی ساخته می‌شود که Resolve شود
    locator.RegisterSingleton<ILogger>([]() {
        std::cout << "Logger created!" << std::endl;
        return std::make_shared<ConsoleLogger>();
    });
    // استفاده
    auto logger1 = locator.Resolve<ILogger>(); // "Logger created!" چاپ می‌شود
    auto logger2 = locator.Resolve<ILogger>(); // دیگر چاپ نمی‌شود (کش شده)
    // Transient - هر بار یک instance جدید
    locator.RegisterTransient<ILogger>([]() {
        return std::make_shared<ConsoleLogger>();
    });

    return 0;
}
```

💡 توجه: در کد بالا در داخل تابع main:

- یا باید در خود تابع main یک شی از نوع ConsoleLogger بسازیم و از متد دوم RegisterSingleton مخصوص نمونه حاضر، رجیستر کنیم تا آن شی را به واسطه ServiceLocator در همه جای برنامه به عنوان یک شی Singleton حاضر در دست داشته باشیم!
- یا باید به واسطه متد اول RegisterSingleton، به صورت تنبل (lazy) رجیستر کنیم تا با پاس دادن تابع factory آن و در اولین استفاده از آن توسط متد Resolve، شی Singleton آن توسط Factory ساخته شود و مورد استفاده همگان واقع شود!
- یا باید به واسطه متد RegisterTransient آنرا رجیستر کنیم تا با پاس دادن تابع Factory آن به ServiceLocator بتوانیم به صورت کثیر انتشار از روی آن، اشیای مختلف بسازیم!

🌸 دو نوع اصلی Service Locator

طبق تعریف اصلی در الگوهای J2EE و بسط داده شده توسط فاولر، Service Locator می‌تواند به دو شکل باشد:

- **Static/Global Service Locator**: از طریق یک متد Static در همه‌جای برنامه در دسترس است (مثال اول). این حالت ساده‌ترین اما بحث برانگیزترین شکل است، چون عملاً یک Global State ایجاد می‌کند.
- **Injected Service Locator**: خود Locator به‌عنوان یک وابستگی (از طریق Constructor) به کلاس‌هایی که به آن نیاز دارند تزریق می‌شود:

```
class OrderProcessor {
private:
    std::shared_ptr<IServiceLocator> locator_;
public:
    explicit OrderProcessor(std::shared_ptr<IServiceLocator> locator)
        : locator_(std::move(locator)) {}

    void Process(const Order& order) {
        auto logger = locator_->Resolve<ILogger>();
        // ...
    }
};
```

نسخه دوم کمی بهتر از نظر تست‌پذیری است (چون می‌توان یک Locator جعلی (Mock) تزریق کرد، اما همچنان مشکل اصلی الگو (که در بخش بعد می‌بینیم) را حل نمی‌کند!

👤 مقایسه دقیق با Dependency Injection:

این مهم‌ترین بخش برای درک درست Service Locator است، چون اغلب با Dependency Injection (DI) اشتباه گرفته می‌شود! با این که هر دو یک هدف مشترک دارند (کاهش وابستگی محکم)، اما مکانیزم متفاوتی دارند.

🎨 مثال اول با Dependency Injection:

```
class OrderProcessor {
public:
    OrderProcessor(std::shared_ptr<ILogger> logger,
                  std::shared_ptr<IEmailService> emailService,
                  std::shared_ptr<IPaymentGateway> paymentGateway)
        : logger_(std::move(logger)),
          emailService_(std::move(emailService)),
          paymentGateway_(std::move(paymentGateway)) {}

    void Process(const Order& order){
        logger_>Log("Processing order...");
        paymentGateway_>Charge(order.total);
        emailService_>SendConfirmation(order.customerEmail);
    }

private:
    std::shared_ptr<ILogger> logger_;
    std::shared_ptr<IEmailService> emailService_;
    std::shared_ptr<IPaymentGateway> paymentGateway_;
};
```

🔗 تفاوت بنیادین: وابستگی‌های آشکار در برابر وابستگی‌های پنهان

اصلی‌ترین تفاوت بین Dependency Injection (DI) و Service Locator در نحوه نمایش وابستگی‌های یک کلاس است. در Dependency Injection، وابستگی‌های کلاس به صورت آشکار (Explicit Dependencies) تعریف می‌شوند؛ یعنی کلاس تمام وابستگی‌های خود را در قالب انتزاع‌ها Interface یا Abstract Class از طریق Constructor (روش استاندارد) یا در موارد خاص از طریق Setter دریافت می‌کند.

تزریق وابستگی از طریق Constructor روش استاندارد محسوب می‌شود، زیرا سازنده کلاس دقیقاً یک بار در هنگام ایجاد شیء اجرا می‌شود و بنابراین می‌توان اطمینان داشت که تمام وابستگی‌های ضروری پیش از استفاده از شیء در اختیار آن قرار گرفته‌اند. در مقابل، هیچ تضمینی وجود ندارد که Setterها حتماً فراخوانی شوند؛ به همین دلیل معمولاً از Setter Injection تنها برای وابستگی‌های اختیاری استفاده می‌شود.

یکی از مزایای Constructor Injection این است که می‌توان در همان Constructor اعتبار وابستگی‌ها را بررسی کرد. برای مثال اگر یکی از وابستگی‌ها مقدار null داشته باشد، کلاس می‌تواند بلافاصله Exception مناسبی پرتاب کند و از ایجاد شیئی که در وضعیت نامعتبر قرار دارد جلوگیری نماید.

در مقابل، در الگوی Service Locator معمولاً کلاس به جای دریافت مستقیم تمام وابستگی‌های خود، تنها به یک Service Locator دسترسی دارد. این Locator ممکن است از طریق Singleton، یک شیء سراسری یا حتی به صورت Static در اختیار کلاس قرار گیرد. سپس هر زمان که کلاس به سرویسی نیاز داشته باشد، آن را از Service Locator درخواست می‌کند.

در هر دو رویکرد، معمولاً وابستگی‌ها به صورت انتزاعی Interface یا Abstract Class مورد استفاده قرار می‌گیرند؛ اما تفاوت در این است که در Dependency Injection وابستگی‌ها از ابتدا در اختیار کلاس قرار می‌گیرند، در حالی که در Service Locator کلاس در زمان نیاز آن‌ها را از Locator درخواست می‌کند.

تفاوت مهم دیگر، نحوه نگهداری وابستگی‌ها است. در Dependency Injection معمولاً وابستگی‌های دریافت‌شده در Property های خصوصی کلاس ذخیره می‌شوند و از همان‌ها استفاده می‌شود. اما در Service Locator الزاماً نیازی به نگهداری آن‌ها وجود ندارد و کلاس می‌تواند هر بار که به سرویسی احتیاج داشت، آن را مجدداً از Locator دریافت کند.

این تفاوت روی نحوه مدیریت خطا نیز اثر می‌گذارد. در Dependency Injection، چون تمام وابستگی‌ها در زمان ساخت شیء مشخص هستند، خطاهای مربوط به نبود وابستگی‌ها معمولاً همان ابتدا شناسایی می‌شوند. اما در Service Locator ممکن است درخواست یک سرویس در بخش‌های مختلف کد انجام شود؛ بنابراین اگر سرویس موردنظر در Locator وجود نداشته باشد، خطا در زمان اجرای همان قسمت از برنامه ظاهر خواهد شد، نه لزوماً هنگام ایجاد شیء. البته اگر کلاس در Constructor تمام سرویس‌های موردنیاز خود را از Service Locator دریافت و اعتبارسنجی کند، این تفاوت تا حد زیادی از بین می‌رود.

از دیدگاه کارایی نیز تفاوت بسیار ناچیزی وجود دارد. در Dependency Injection معمولاً کلاس مستقیماً به مرجع (Reference) وابستگی خود دسترسی دارد، در حالی که در Service Locator هر بار لازم است سرویس از یک ساختار داده مانند Hash Table یا Map بازیابی شود. اگرچه از نظر تئوری دسترسی مستقیم سریع‌تر است، اما در اکثر کاربردهای واقعی این اختلاف آن‌قدر ناچیز است که در انتخاب بین این دو الگو نقش تعیین‌کننده‌ای ندارد.

یکی از مزایایی که معمولاً برای Service Locator مطرح می‌شود، متمرکز بودن دسترسی به سرویس‌ها است. از آنجا که تمام کلاس‌ها وابستگی‌های خود را از یک نقطه مشترک دریافت می‌کنند، این نقطه می‌تواند نحوه ایجاد، نگهداری و بازگرداندن سرویس‌ها را مدیریت کند. برای مثال، اگر سیاست سیستم بر این باشد که یک سرویس همواره باید در دسترس باشد، Locator یا Container می‌تواند در صورت حذف شدن یا از بین رفتن نمونه قبلی، نمونه جدیدی ایجاد کرده و آن را در اختیار درخواست‌کننده قرار دهد. در نتیجه تمام مصرف‌کنندگان بدون اطلاع از این فرآیند، همچنان به سرویس معتبر دسترسی خواهند داشت.

البته باید توجه داشت که این قابلیت بیشتر به IoC Container و مدیریت چرخه عمر (Lifetime Management) مربوط است تا ذات الگوی Service Locator! همچنین همین قابلیت را می‌توان در بسیاری از پیاده‌سازی‌های Dependency Injection نیز توسط Container های مدرن مشاهده کرد.

در نهایت، مهم‌ترین مزیت Dependency Injection این است که وابستگی‌های هر کلاس کاملاً آشکار هستند و تنها با مشاهده Constructor می‌توان فهمید که کلاس برای انجام وظیفه خود به چه سرویس‌هایی نیاز دارد. در مقابل، در Service Locator وابستگی‌ها درون بدنه کلاس پنهان می‌شوند و تا زمانی که پیاده‌سازی کلاس بررسی نشود، مشخص نیست که کلاس از چه سرویس‌هایی استفاده می‌کند. به همین دلیل بسیاری از متخصصان Service Locator را نوعی Anti-Pattern می‌دانند، زیرا وابستگی‌های واقعی کلاس را مخفی می‌کند و خوانایی، تست‌پذیری و قابلیت نگهداری سیستم را کاهش می‌دهد.

```
class OrderProcessor {
public:
    OrderProcessor() = default;    // Appears to have no dependencies

    void Process(const Order& order) {
        auto gateway =
            ServiceLocator::Resolve<std::shared_ptr<IPaymentGateway>>();
        // Surprise! The dependency is discovered only
        // by reading the implementation of Process().
    }
};
```

مارک سیمن این مشکل را وابستگی دروغین (Dependency Lying) می‌نامد! کلاس با داشتن یک Constructor بدون پارامتر، به مصرف‌کننده دروغ می‌گوید که «من به چیزی نیاز ندارم»، در حالی که در واقعیت به چند سرویس داخلی وابسته است که فقط با خواندن کامل کد کشف می‌شوند!

 **مناظره تاریخی: آیا Service Locator یک ضدالگو (Anti-Pattern) است؟**

 **دیدگاه مارک سیمن: «Service Locator is an Anti-Pattern»**

در سال ۲۰۱۰، مارک سیمن مقاله‌ای بسیار تأثیرگذار با همین عنوان منتشر کرد. استدلال اصلی او:

1- **پنهان‌سازی وابستگی‌ها API را گمراه‌کننده می‌کند:** کسی که یک کلاس را مصرف می‌کند، از طریق Constructor باید بتواند دقیقاً بفهمد این کلاس به چه چیزهایی نیاز دارد. Service Locator این قرارداد را می‌شکند!

2- **مشکلات تست عمیق‌تر از آنچه به نظر می‌رسد هستند:** وقتی وابستگی‌ها از طریق یک Global Locator تأمین می‌شوند، تنظیم صحیح حالت آن Locator قبل از هر تست (و Reset کردنش بعد از هر تست، برای جلوگیری از نشت وضعیت بین تست‌ها) خودش منبع پیچیدگی و باگ می‌شود.

3- **خطاها دیرتر کشف می‌شوند:** با DI، اگر یک وابستگی حیاتی فراموش شود، برنامه اصلاً Wiring نمی‌شود و خطا زود (در Composition Root) ظاهر می‌شود. با Service Locator، ممکن است خطا فقط وقتی رخ دهد که یک مسیر خاص از کد در Production اجرا شود که می‌تواند بسیار دیرتر و خطرناک‌تر باشد.

 **دیدگاه مارتین فاولر: نگاه متعادل‌تر**

فاولر در مقاله معروف خود یعنی Inversion of Control Containers and the Dependency Injection Pattern در (۲۰۰۴) هر دو الگو (Service Locator و Dependency Injection) را به‌عنوان دو راه‌حل معتبر برای یک مسئله مشترک (Inversion of Control) معرفی می‌کند، بدون این‌که Service Locator را به‌طور کامل رد کند.

دیدگاه او: «انتخاب بین Service Locator و Dependency Injection تا حد زیادی موضوعی جزئی است.» او تأکید می‌کند مسئله اصلی، خود اصل Inversion of Control است، نه این‌که کدام مکانیزم دقیق برای پیاده‌سازی آن استفاده شود.

او می‌پذیرد که Dependency Injection معمولاً وابستگی‌ها را شفاف‌تر نشان می‌دهد، اما Service Locator را در برخی زمینه‌ها (مثلاً وقتی کلاسی که وابستگی می‌گیرد از یک فریم‌ورک خارجی می‌آید که کنترل Constructor آن دست شما نیست) کاربردی می‌داند.

نتیجه این که، دیدگاه رایج امروز صنعت به این شکل تثبیت شده: Dependency Injection به عنوان رویکرد پیش فرض و ترجیحی و Service Locator به عنوان ابزاری برای موارد خاص و استثنایی، نه یک جایگزین عمومی برای DI موضوعیت دارند!

✓ چه زمانی Service Locator هنوز کاربرد مشروع دارد؟

با وجود انتقادات جدی، چند سناریو وجود دارد که Service Locator (یا الگوهای مشابه آن) همچنان توجیه پذیر است:

1- زمانی که Constructor تحت کنترل شما نیست: بعضی فریم ورک ها (به خصوص در .NET Framework. قدیمی یا کتابخانه های legacy) نمونه سازی برخی کلاس ها را (مثل Attribute ها یا برخی کلاس های خاص ASP.NET قدیمی) خودشان مدیریت می کنند و شما امکان تزریق وابستگی از طریق Constructor را ندارید. در این موارد، دسترسی به یک Service Locator محدود، گاهی تنها راه عملی است.

```
class AuditFilter {
public:
    void BeforeExecute(ActionContext& context) {
        // The framework owns the lifetime of this object.
        auto logger =
            context.Services().Resolve<std::shared_ptr<IAuditLogger>>();

        logger->Log(context.ActionName());
    }
};
```

جالب است بدانید حتی این استفاده هم (که در بسیاری از فریم ورک های مدرن مثل ASP.NET Core به نام Service Locator داخلی از طریق ServiceProvider وجود دارد) دقیقاً همان الگوی بحث برانگیز است، منتها استفاده از آن محدود به لبه های فریم ورک (Framework Boundary) شده، نه در قلب منطق کسب و کار.

2- کتابخانه های افزونه محور (Plugin Architecture): وقتی یک سیستم باید افزونه هایی را که در زمان کامپایل شناخته شده نیستند، به صورت پویا بارگذاری کند، معمولاً نوعی Registry/Locator برای یافتن سرویس های مربوط به هر افزونه لازم است، چون نمی توان از قبل همه وابستگی ها را در Constructor اعلام کرد.

3- دسترسی اختیاری به سرویس های Cross-Cutting (با احتیاط): برخی سرویس های سراسری مثل Logger گاهی از طریق یک الگوی مشابه Service Locator (مثل Log.Logger در Serilog) در دسترس قرار می گیرند. این کار پذیرفته شده تر است چون Logger معمولاً یک وابستگی «نگرانی جانبی» (Cross-Cutting Concern) است، نه بخشی از منطق کسب و کار اصلی، و معمولاً هیچ رفتار متفاوتی بین تست و Production ندارد که باعث دردرس شود.

💡 راهکار میانه: از Service Locator به سمت DI Container

نکته ای که اغلب باعث سردرگمی می شود: DI Container (مثل Autofac، Microsoft.Extensions.DependencyInjection، یا Ninject) با Service Locator یکی نیست، هرچند از نظر ظاهری شبیه به هم به نظر می رسند (هر دو یک Registry مرکزی از سرویس ها دارند).


```

#include <iostream>
#include <memory>
#include <unordered_map>
#include <typeindex>
// ===== Interface =====
class IPaymentGateway {
public:
    virtual void Charge(double amount) = 0;
    virtual ~IPaymentGateway() = default;
};
// ===== Implementation =====
class StripeGateway : public IPaymentGateway {
public:
    void Charge(double amount) override{
        std::cout << "Charged: " << amount << std::endl;
    }
};
// ===== Order =====
class Order {
public:
    double Total = 100.0;
};
// ===== استفاده به شکل Service Locator که ضد الگوست =====
// =====
class ServiceProvider {
private:
    std::unordered_map<std::type_index, std::shared_ptr<void>> services;
public:
    template<typename T>
    void AddService(std::shared_ptr<T> service) {
        services[typeid(T)] = service;
    }

    template<typename T>
    std::shared_ptr<T> GetService(){
        return std::static_pointer_cast<T>(
            services[typeid(T)]
        );
    }
};

class BadOrderProcessor{
private:
    ServiceProvider* _serviceProvider;
public:
    BadOrderProcessor(ServiceProvider* provider)
        : _serviceProvider(provider){}
    void Process(Order order){
        // کلاس خودش از Container درخواست می‌کند
        // Service Locator همان مشکل
        auto gateway = _serviceProvider->GetService<IPaymentGateway>();
        gateway->Charge(order.Total);
    }
};

```

```
// =====
// Composition Root فقط DI Container صحیح استفاده =====
// =====
class OrderProcessor {
private:
    std::shared_ptr<IPaymentGateway> _gateway;
public:
    // وابستگی از بیرون تزریق می‌شود
    // بدون اینکه OrderProcessor چیزی درباره Container بداند
    OrderProcessor(std::shared_ptr<IPaymentGateway> gateway)
        : _gateway(gateway) {}

    void Process(Order order) {
        _gateway->Charge(order.Total);
    }
};
// =====
// ===== Composition Root =====
// =====
int main()
{
    ServiceProvider provider;
    // مشابه:
    // services.AddSingleton<IPaymentGateway, StripeGateway>();
    auto gateway = std::make_shared<StripeGateway>();
    provider.AddService<IPaymentGateway>(gateway);
    // -----
    // روش غلط Service Locator
    // -----
    BadOrderProcessor badProcessor(&provider);
    badProcessor.Process(Order());
    // -----
    // روش صحیح Dependency Injection
    // -----
    // فقط اینجا استفاده می‌شود
    auto processor = OrderProcessor(
        provider.GetService<IPaymentGateway>());
    processor.Process(Order());
    return 0;
}
```

 **قانون طلایی:** خود ابزار (DI Container) مشکل نیست؛ مشکل جایی است که از آن در همه‌جای کد به‌جای فقط در Composition Root (نقطه واحدی که کل گراف وابستگی‌ها ساخته می‌شود، معمولاً نزدیک Main) استفاده شود. اگر یک DI Container فقط در Composition Root استفاده شود و بقیه کلاس‌ها صرفاً از طریق Constructor وابستگی بگیرند، این عملاً Dependency Injection صحیح است، نه Service Locator! حتی اگر از همان ابزار DI Container برای مقدار دهی به Constructor آنها استفاده شده باشد.

جمع‌بندی

Service Locator یک الگوی تاریخی و مهم است که نقش بزرگی در معرفی مفهوم Inversion of Control به دنیای نرم‌افزار سازمانی داشت، اما تجربه صنعت در طول دهه گذشته نشان داده که پنهان‌سازی وابستگی‌ها که ذات این الگو است، هزینه‌های واقعی روی خوانایی، تست‌پذیری، و اطمینان از صحت سیستم دارد. به همین دلیل، امروز اجماع گسترده‌ای در صنعت وجود دارد که معتقدند:

- Dependency Injection باید رویکرد پیش‌فرض برای مدیریت وابستگی‌ها در منطق کسب و کار باشد.

- **Service Locator** باید محدود به موارد استثنایی شود یعنی جایی که واقعاً کنترل روی **Constructor** وجود ندارد (مثل لبه‌های فریم‌ورک یا معماری‌های افزونه‌محور)، نه به عنوان یک الگوی عمومی برای راحتی دسترسی به سرویس‌ها.
 - حتی وقتی از یک **DI Container** استفاده می‌کنید، باید مراقب باشید که آن را به یک **Service Locator** سراسری تبدیل نکنید! کلید اصلی، محدود کردن استفاده از **Container** به یک نقطه واحد (**Composition Root**) است.
- درک این تفاوت ظریف اما حیاتی، دقیقاً همان چیزی است که مناظره سیمن-فاولر را شکل داده و امروز به یکی از اصول پذیرفته‌شده طراحی نرم‌افزار مدرن تبدیل شده است.

○ **Unit of Work**: از انواع الگوهای سازمانی یا Enterprise است و هماهنگ سازی تغییرات داده‌ها به صورت اتمیک را میسر می‌کند!



در دنیای توسعه‌ی نرم‌افزار به ویژه در سیستم‌هایی که با پایگاه داده کار می‌کنند، یکی از چالش‌های اصلی مدیریت تراکنش‌های داده‌ای به‌صورت ایمن و سازگار است. چه می‌شد اگر بتوانیم مجموعه‌ای از تغییرات را به‌صورت یک واحد منطقی ثبت کنیم، به‌طوری که یا همه موفق شوند یا هیچ‌کدام اعمال نشوند؟

اینجاست که پترن Unit of Work (UoW) به کمک ما می‌آید. این پترن یک الگوی سازمانی است که مسئولیت پیگیری، مدیریت و اعمال تغییرات داده‌ها در یک تراکنش واحد را بر عهده دارد. این پترن به‌طور گسترده در معماری‌هایی مانند DDD و Clean Architecture استفاده می‌شود. این همان مفهومی است که در علوم پایگاه داده به تراکنش یا Transaction معروف است! فرض کنید در یک عملیات، می‌خواهید همراه با ثبت یک سفارش، عملیات ثبت رکورد پرداخت بانکی و ثبت فاکتور را هم انجام دهید، به گونه‌ای که یا هر سه ثبت شوند یا هیچ‌کدام ثبت نشوند! مسلماً در پایگاه داده، از Transaction برای این هدف استفاده می‌کنید!

هدف اصلی پترن Unit of Work چیست؟

- جمع‌آوری تمام تغییراتی که در طول یک عملیات منطقی روی اشیاء دامنه (Domain Objects) اعمال شده‌اند.
- ثبت آن تغییرات در یک تراکنش اتمیک (همه یا هیچ!)
- جلوگیری از ذخیره‌سازی اشتباه یا ناقص داده‌ها.
- کاهش فراخوانی‌های مکرر پایگاه داده با دسته‌بندی (batch) کردن عملیات‌ها.

مفاهیم کلیدی:

- 1- پیگیری یا **UoW: Tracking** تمام اشیایی را که درج (Add)، بروزرسانی (Update) یا حذف (Delete) شده‌اند، ردیابی می‌کند.
- 2- **Commit**: زمانی که عملیات منطقی تمام شد، UoW تمام تغییرات را در یک تراکنش واحد به پایگاه داده اعمال می‌کند.
- 3- **Rollback**: در صورت بروز خطا، تمام تغییرات لغو می‌شوند (معمولاً با استفاده از تراکنش‌های پایگاه داده).
- 4- **UoW: Repository Integration**: معمولاً با الگوی Repository همراه است! Repository ها تغییرات را به UoW گزارش می‌دهند، نه مستقیماً به پایگاه داده.

ساختار کلی پترن Unit of Work

در Unit of Work سه لیست داریم:

- چیزهایی که باید درج شوند
- چیزهایی که باید بروزرسانی شوند
- چیزهایی که باید حذف شوند

نسخه ساده‌شده‌ی معماری Unit Of Work

```
Service → Repository → UnitOfWork → Database
```

هدف اصلی این الگو، مدیریت یکپارچه‌ی تغییرات و اعمال آن‌ها به صورت اتمیک (همه یا هیچ) است.

این بخش از کتاب به صورت تکاملی نوشته شده؛ یعنی از مسئله‌ی اولیه شروع می‌کنیم و قدم‌به‌قدم به یک پیاده‌سازی عملی با File System در ++C می‌رسیم.

✗ **مسئله‌ی اصلی (Problem Statement):** فرض کنید در یک UseCase چند عملیات داریم:

- اضافه کردن داده
- ویرایش داده
- حذف داده

اگر هر کدام از این عملیات‌ها مستقلاً روی منبع ذخیره‌سازی اعمال شوند:

- در صورت بروز خطا، سیستم در وضعیت ناسازگار قرار می‌گیرد!
- کنترل تراکنش در کل کد پخش می‌شود!
- تست و نگهداری سخت می‌شود!

راه‌حل ساده ولی ناقص: Transaction دستی

```
beginTransaction();
repo.saveA();
repo.saveB();
commit();
```

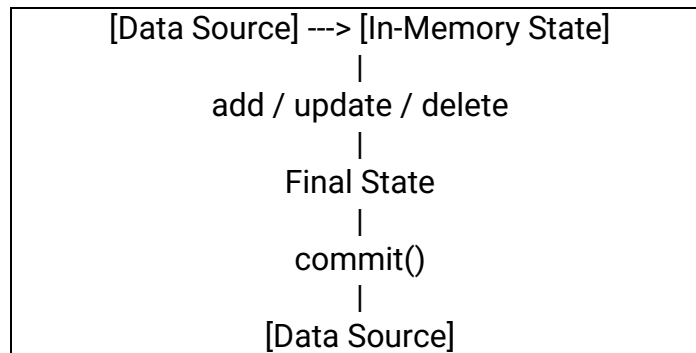
✗ **مشکلات:**

- Business Logic به Persistence وابسته می‌شود! یعنی کدهای مربوط به منطق اصلی برنامه (Business Logic) مستقیماً به نحوه ذخیره‌سازی داده‌ها (Persistence) وابسته می‌شوند.
- تکرار کد زیاد می‌شود!
- تست‌پذیری کاهش می‌یابد!

💡 **ایده Unit of Work:** به‌جای اعمال فوری تغییرات، آن‌ها را جمع کن و در پایان یک‌جا اعمال کن!

- تغییرات را ردیابی می‌کند!
- یک نقطه‌ی واحد برای commit() دارد!
- امکان rollback() منطقی را فراهم می‌کند!

🏠 مدل ذهنی Unit of Work به شکل زیر است:



تا قبل از **Commit** : منبع اصلی حقیقت روی حافظه است و هیچ تغییری روی فایل/دیتابیس اعمال نشده است!

🧩 مثال عملی: کار با File System در C++

فرمت فایل:

1,Ali
2,Sara
3,Reza

حال می‌خواهیم:

- کاربر جدید اضافه کنیم!
- کاربر را ویرایش کنیم!
- کاربر را حذف کنیم!
- و نهایتاً همه‌ی این‌ها با هم و فقط در آخر ذخیره شوند.

Entity

```
struct User {  
    int id;  
    std::string name;  
};
```

مرحله 2 Repository : این بخش فقط با سیستم فایل کار می‌کند! از طرفی Repository عملیات Commit را انجام نمی‌دهد و

فقط عملیات واقعی روی فایل را بلد است!

Repository (فقط I/O)

```
class UserRepository {  
public:  
    explicit UserRepository(const std::string& file)  
        : filename(file) {}  
    std::vector<User> loadAll() {  
        std::vector<User> users;  
        std::ifstream in(filename);  
        std::string line;  
        while (std::getline(in, line)) {  
            auto comma = line.find(',');  
            users.push_back({  
                std::stoi(line.substr(0, comma)),  
                line.substr(comma + 1)  
            });  
        }  
        return users;  
    }  
};
```

```

void saveAll(const std::vector<User>& users) {
    std::ofstream out(filename, std::ios::trunc);
    for (auto& u : users) {
        out << u.id << "," << u.name << "\n";
    }
}
private:
    std::string filename;
};

```

: Repository 📦

- چیزی از Unit of Work نمی‌داند!
- فقط خواندن و نوشتن را بلد است!

: Unit of Work 🌱 پیاده سازی اصلی

```

class UnitOfWork {
public:
    explicit UnitOfWork(UserRepository& repo)
        : repo(repo), users(repo.loadAll()) {}
void registerNew(const User& user); // افزودن کاربر موقت جدید
void registerDirty(const User& user); // ویرایش روی داده های موقت
void registerRemoved(int userId); // درخواست حذف کاربر وارد این تابع می شود

void commit();
void rollback();
private:
    UserRepository& repo; //user repository
    std::vector<User> users; // snapshot
    std::vector<User> newUsers; //new users tmp list
    std::vector<User> dirtyUsers; //edited users tmp list
    std::vector<int> removedIds; //deleted users tmp list
};

```

:Commit() تابع 🌱

```

void UnitOfWork::commit() {
    // apply inserts
    // apply updates
    // apply removals
    repo.saveAll(users);
    clear();
}

```

:rollback() تابع 🌱

```

void UnitOfWork::rollback() {
    newUsers.clear();
    dirtyUsers.clear();
    removedIds.clear(); // users vector remains unchanged
}

```

Rollback یعنی هیچ چیزی Persist نمی شود و حافظه همان Snapshot اولیه می ماند!

```

void updateUser(UnitOfWork& uow) {
    uow.registerNew({4, "Hadi"}); // هادی به عنوان کاربر چهارم به فایل اضافه خواهد شد
    uow.registerDirty({2, "Sara Updated name"}); // نام سارا تغییر کرد
    uow.registerRemoved(1); // علی حذف خواهد شد
    uow.commit();
}

```


چرا برای ویرایشات از لیستی با نام Dirty یعنی کثیف استفاده می شود: Dirty یا کثیف یعنی این آبجکت از زمانی که از دیتابیس خوانده شده، تغییر کرده و دیگر با نسخه ذخیره شده یکی نیست.

🚀 نکات پیشرفته تکمیلی: ایده RAIL-based Unit of Work

```
class UnitOfWork {
public:
    ~UnitOfWork() {
        if (!committed)
            rollback();
    }
};
```

اگر commit فراموش شود rollback خودکار ران می شود!

📚 توضیح خلاصه RAIL

RAIL یا Resource Acquisition Is Initialization یعنی عمر یک منبع را به عمر یک شیء گره بزن! منبعی مثل حافظه، فایل، قفل یا اتصال را هنگام ساختن شیء بگیر و در destructor همان شیء آزادش کن؛ بنابراین حتی اگر وسط کار exception رخ بدهد، با از بین رفتن شیء منبع هم به طور خودکار آزاد می شود. خلاصه: ساخت شیء = گرفتن منبع، نابودی شیء = آزاد کردن منبع.

✂️ Delta-based UoW: به جای snapshot کامل:

- فقط تغییرات ذخیره می شوند!
- مناسب دیتابیس های بزرگ!

✂️ نسخه تکمیلی Unit Of Work ما:

```
class UnitOfWork {
public:
    void registerNew(Entity* e) {
        newEntities.push_back(e);
    }

    void registerDirty(Entity* e) {
        dirtyEntities.push_back(e);
    }

    void registerRemoved(Entity* e) {
        removedEntities.push_back(e);
    }

    void commit() {
        beginTransaction();

        for (auto e : newEntities)
            e->insert();

        for (auto e : dirtyEntities)
            e->update();
    }
};
```

```

        for (auto e : removedEntities)
            e->remove();

        commitTransaction();
    }
private:
    std::vector<Entity*> newEntities;
    std::vector<Entity*> dirtyEntities;
    std::vector<Entity*> removedEntities;
};

```

نسخه کامل تر: 

```

class UnitOfWork {
public:
    explicit UnitOfWork(UserRepository& repo)
        : repo(repo), users(repo.loadAll()) {}
    // اضافه کردن یوزر
    void registerNew(const User& user) {
        newUsers.push_back(user);
    }
    // ویرایش یوزر
    void registerDirty(const User& user) {
        dirtyUsers.push_back(user);
    }
    // حذف یوزر
    void registerRemoved(int userId) {
        removedUserIds.push_back(userId);
    }

    void commit() {
        // اضافه
        for (auto& nu : newUsers) {
            users.push_back(nu);
        }
        // ویرایش
        for (auto& du : dirtyUsers) {
            for (auto& u : users) {
                if (u.id == du.id) {
                    u.name = du.name;
                    break;
                }
            }
        }
        // حذف
        auto it = std::remove_if(//لیست حذف شده هاست
            users.begin(),
            users.end(),
            [&](const User& u) {
                return std::find(
                    removedUserIds.begin(),
                    removedUserIds.end(),
                    u.id) != removedUserIds.end();
            });
        users.erase(it, users.end());
    }
private:

```

```

void clear() {
    newUsers.clear();
    dirtyUsers.clear();
    removedUserIds.clear();
}

UserRepository& repo;

std::vector<User> users;
std::vector<User> newUsers;
std::vector<User> dirtyUsers;
std::vector<int> removedUserIds;
};

```

📁 در این ساختار:

Repository: مسئول دسترسی و عملیات مستقیم روی داده‌هاست.

Unit of Work: تغییرات را ثبت می‌کند و در `commit()` آن‌ها را یکجا اعمال می‌کند.

✅ مزایای **Unit Of Work**:

- تست راحت‌تر
- کنترل تراکنش مرکزی
- آماده برای DDD و CQRS و Event Sourcing

📌 جمع‌بندی نهایی:

Unit Of Work یعنی ساختن آخرین نسخه معتبر داده‌ها در حافظه و **Persist** کردن آن به صورت اتمیک!

همچنان که مشاهده فرمودید، با دریافت آیتم‌های تازه اضافه شده، آیتم‌های تغییر یافته و آی دی آیتم‌های پاک شده به ترتیب زمانی، و نهایتاً اضافه کردن آیتم‌های جدید اضافه شده و سپس ویرایش آیتم‌های ویرایش شده و سپس پاک کردن آیتم‌های با آی دی‌های پاک شده، بسیاری از مشکلات ناسازگاری در داده حل می‌شود! شاید سوالی بوجود آمده باشد که اگر در حین عملیات **Commit** مشکلی رخ داد چه؟ مثلاً اگر برق قطع شد یا دیسک خراب شد یا ...؟ فرض کنید گروهی از رکوردها به روز شدند و ناگهان مشکل رخ داد! پاسخ: الگوی ذخیره سازی همان **Unit Of Work** باشد! ذخیره فایل را روی نامی موقت انجام دهید و سپس با دستور اتمیک **Move** در سیستم فایل (همان **Rename** فایلی)، فایل موقت را جایگزین فایل اصلی کنید چون عملیات **Rename** یا همان **Move** عملیات اتمیک است و یا اتفاق می‌افتد و یا کاملاً کنسل می‌شود! به هر حال الگوی **Unit Of Work** الگوی پایه **ORM** ها و **DDD** و **CQRS** است!

فصل پنجم

الگوهای معماری (Architectural Patterns)

الگوهای معماری، بیش از آنکه راهکاری استاندارد و تست شده جهت حل مشکلی خاص باشند، ساختاری استاندارد برای استخوان بندی یک نرم افزار هستند و تفاوت این نوع الگوها با الگوهای طراحی (Design Patterns)، در نوع استفاده از آنها است! به بیان بهتر دیزاین پترن ها یا الگوهای طراحی برای حل مشکل خاصی به کار می روند، ولی الگوهای معماری، قالب بندی کلی یک نرم افزار را تعریف می کنند!

○ **MVC (Model-View-Controller):** این الگوی معماری، جزو الگوهای معماری لایه پرزنتیشن (Presentation Architectural pattern) است و بیشتر در نرم افزارهایی که **View** قابل نمایش سمت کلاینت دارند مورد استفاده قرار می گیرد و در واقع می توان این الگوی معماری را جزو معروف ترین و همه گیرترین انواع معماری نرم افزاری دانست که در درصد بالایی از نرم افزارهای تولید شده دنیا پیاده شده اند!

در این معماری ما با سه لایه سر و کار داریم:

● **Model:** این کلاس، بانک متغیرهای **private** از نوع **Primitive** (int, bool, string, double, enum, ...) و یا **Struct** های متشکل از متغیرهای **Primitive** است! به بیان بهتر ما تمام متغیرهای زیر سیستم **MVC** خود را در لایه **Model** نگه می داریم! پس این **Property** های **private** باید **Getter** و **Setter** اختصاصی خودشان را داشته باشند، و در **Setter** هایشان هم شرط می گذاریم که چنانچه مقدار ورودی **Setter** با مقدار قبلی آن **Property** برابر نباشد، مقدار جدید در آن **Property** تنظیم شود و یک نوتیفیکیشن (معمولا با استفاده از سیگنال در **Qt** یا **Boost** یا توسط یک **Callback** یا توسط الگوی طراحی **Observer**) از شی ساخته شده از کلاس **Model** منتشر شود! شایان ذکر است این کلاس هیچ دسترسی به **View** و **Controller** ندارد و آنها را هم نمی شناسد! در هنگام استفاده هم یک نمونه شی از آن ساخته می شود و به صورت اشاره گر یا رفرنس، به **Controller** و **View** پاس داده می شود! در زمینه صحت سنجی هم می توان یک **Setter** را در این لایه، مجهز به صحت سنجی داده کرد تا داده غیر مجاز را تنظیم نکند!

● **View:** این لایه، لایه نمایش کاربری با **GUI** است و حاوی دستورات نمایش فرم و اطلاعات سمعی و بصری برای کاربر می باشد! پس این لایه مختص تعامل و ارتباط معنایی با کاربر نرم افزار است! معمولا در **Framework** های مختلف، این لایه ترکیبی از دو نوع فایل است:

1- یک فایل **XML** یا **Json** یا **QML** یا ... که نقش **GUI** را بازی می کند و مستقیما آیتم های گرافیکی همچون **Button** و **ListView** را داخل آنها تعریف می کنیم و به هر کدامشان **id** و **Position** و **Scale** و **Rotation** و ... تنظیم می کنیم! برای مثال دستورات **QML** یا دستورات ساخت رابط کاربری و صفحات **html** و **JS** و ... و یا فایل های **ui** در **QWidget** که با باز کردنش در **Text Editor** ها، دستورات **XML** داخل آنها به نمایش در می آید و یا همچنین فایل های **xml** فرم های دات نت!

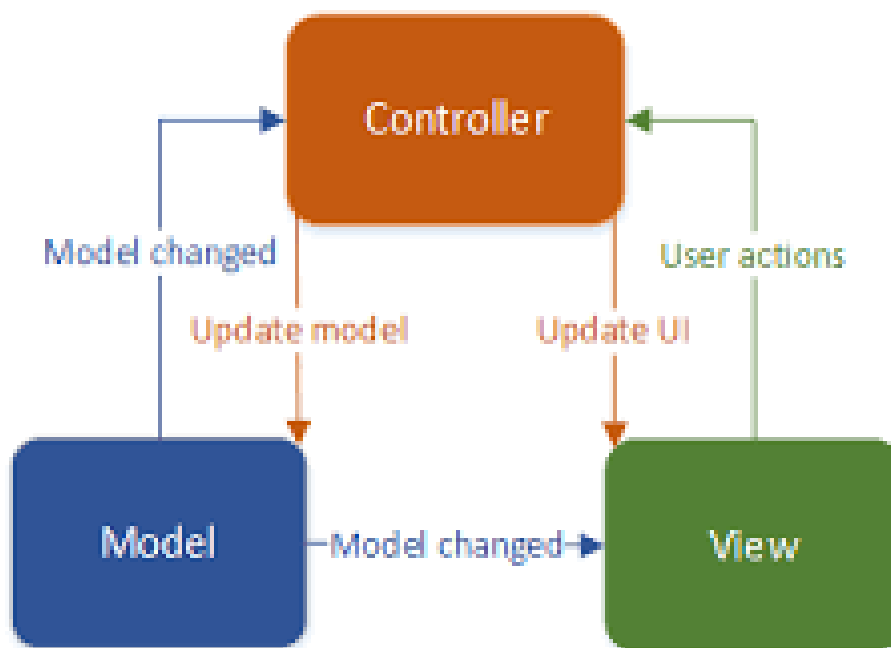
2- کلاس موازی با فایل **XML** و ... که معمولا مستقیما دستورات زبان برنامه نویسی را داخل آن می نویسید! پس در این کلاس می توانید با استفاده از **id** اقلام فایل **GUI** به آیتم های گرافیکی دسترسی پیدا کنید و برای آنها **Event** و ... تنظیم کنید و مثلا دستور **Click** یک دکمه را به یک تابع در داخل این کلاس **Bind** کنید! البته گاهی لازم می شود صحت اطلاعات ورودی کاربر در این لایه نیز بررسی شده و سپس به لایه میانجی ارسال شود (بخشی از وظیفه دو لایه دیگر)! برای مثال اگر قرار نیست یک **Input Field** مقادیر حروف دار را از کاربر دریافت کند، با استفاده از **Regular Expression** می توانیم به آن **Input Text** بفهمانیم که فقط ورودی اعداد یا الگویی خاص چون شماره تلفن یا ایمیل با فرمت مناسب را دریافت کند. در این لایه، تغییرات **Model** توسط نوتیفیکیشن به صورت خود به خود در نمایش اطلاعات به کاربران، آپدیت می شود ولی برای اعمال تغییرات در دیتای مدل، لایه **View** به **Controller** مراجعه می کند و با استفاده از

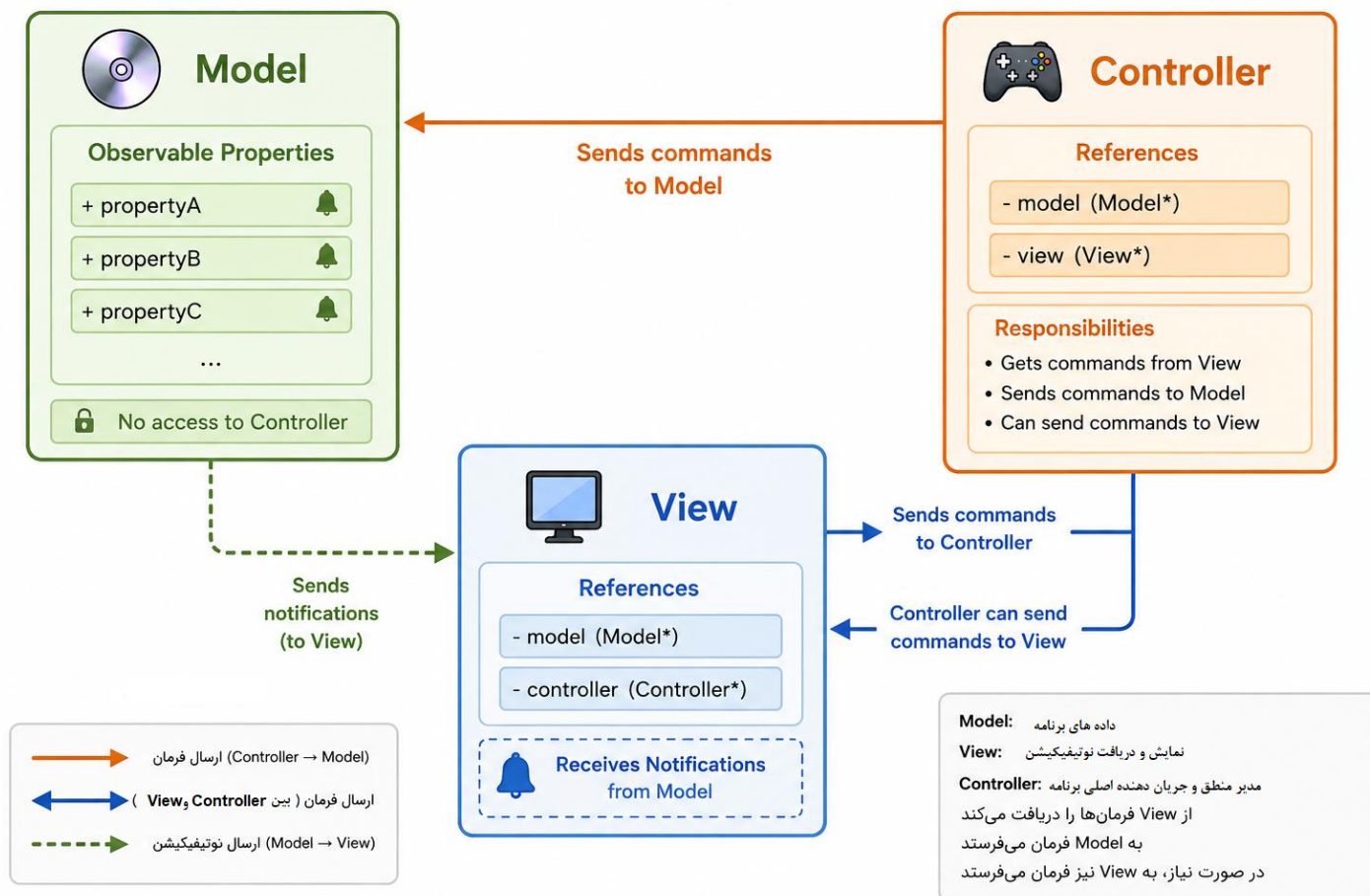
متدی خاص، ویرایش اطلاعات را درخواست می دهد تا Controller بنا به مصلحت، دیتای صحیح را در Model تنظیم نماید! قطعا به شرطی که Controller، ویرایش اطلاعات داخل Model را تایید کند و انجام دهد، نوتیفیکیشن های Model، تغییرات جدید را منتشر می کنند تا نمایش اطلاعات جدید و لازم، در View به روز رسانی شوند! این لایه یک اشاره گر (رفرنس) به Model و یک اشاره گر به Controller دارد!

Controller: این لایه، لایه محاسبات و منطق و مدیریت و الگوریتم و عمل است! این لایه به هر دو لایه Model و View دسترسی دارد! پس توابع این لایه، به واسطه خودش یا View به اصطلاح Call می شود تا محاسباتی را انجام دهد و یا تغییراتی در مدل به صلاحدید خودش انجام دهد! این لایه یک شی ساخته شده از کلاس Controller است که داخلش هیچ متغیری ندارد جز یک اشاره گر به Model اش و یک اشاره گر به View اش (می تواند چند اشاره گر به چند فرم پیوسته داشته باشد) و اگر نیاز به Controller دیگری مانند Controller دسترسی به DataBase یا Controller ارسال و دریافت داده به / از Web Server را داشته باشد، اشاره گر به آنها را نیز خواهد داشت! پس شامل اشیایی از کنترلرهای دیگر یا شی مدل و View خودش می باشد و شامل متغیرهای primitive نمی باشد بلکه متغیرهای primitive داخل شی مدل قرار دارند!

به طور خلاصه: اگر قرار است یک گروه از محاسبات سبک یا سنگین با الگوریتم های ساده و یا پیشرفته انجام شود، در لایه Controller انجام می شود! Controller به هر دو لایه Model و View دسترسی مستقیم دارد. Controller ویرایشات مستقیم را به شرط صلاحدیدش در Model انجام می دهد و سپس تغییرات Model، از داخل خود Model نوتیفای می شود و View توسط نوتیفای Model، نمایش اطلاعاتش را به روز می کند! View هم به Controller و هم به Model دسترسی دارد ولی تغییرات مورد نیاز در مدل را به واسطه و با صلاحدید Controller اعمال می کند و خودش مستقیما در ویرایش اطلاعات Model دخالت نمی کند! Model به هیچ کدام از لایه های دیگر دسترسی ندارد ولی Controller و View به هر دو لایه دسترسی دارند!

MVC





✂️ دیزاین پترن های استفاده شده در معماری MVC:

- 1- در لایه Model به جهت اینکه باید نوتیفیکیشن تغییرات مدل به سمت View برود تا View هم به فراخور تغییرات در Model، آیتم های نمایشی خود را به روز کند باید از Observer به جهت ماهیت Event Base بودنش استفاده کند!
- 2- در Controller به جهت اینکه ماهیت پردازشی و تصمیم گیری و بازیابی داده را دارد، از انواع دیزاین پترن ها می تواند استفاده کند ولی اصلی ترین پترنی که به نظر می رسد در کنترلر بسیار موثر است، پترن Strategy است! علل الخصوص که شاید چندین کنترلر برای یک بیزنس در Context های مختلف داشته باشیم (نظیر ریپازیتوری دیتابیس در انواع دیتابیس ها یا فایل سیستم ها) که در این شرایط می توانیم برای هر مورد کنترلر، یک کلاس جداگانه طراحی کنیم که این همان الگوی استراتژی می باشد!
- 3- همیشه در لایه View که همان لایه GUI / UI برنامه های مختلف است، ساختار Parent-Child بین آیتم های UI وجود دارد که این ما را سمت الگوی Composite سوق می دهد!

```
#include <iostream>
#include <functional>
#include <string>
// =====
// Forward declaration
// =====
class Controller;
```

```

// =====
// Model
// =====
class Model {
private:
    std::string name;
    int age;

    std::function<void()> onChanged; // callback for notify (pointer to function)

public:
    Model(const std::string& name, int age)
        : name(name), age(age) {}

    // -----
    // Getters
    // -----
    const std::string& getName() const {
        return name;
    }

    int getAge() const {
        return age;
    }

    // -----
    // Setters
    // -----
    void setName(const std::string& newName) {
        if (name == newName) {
            return;
        }

        name = newName;
        notify();
    }

    void setAge(int newAge) {
        if (age == newAge) {
            return;
        }

        age = newAge;
        notify();
    }

    // -----
    // Observer for notify
    // -----
    void setOnChanged(std::function<void()> callback) {
        onChanged = callback;
    }

private:
    void notify() {
        if (onChanged) {
            onChanged();
        }
    }
};

```

```

// =====
// View
// =====
class View {
private:
    Model* model;
    Controller* controller;

    std::string displayedName;
    int displayedAge;

public:
    View(Model* model)
        : model(model),
          controller(nullptr),
          displayedName(model->getName()),
          displayedAge(model->getAge()) {

        model->setOnChanged( //pass my callback for notification
            [this]() {
                onModelChanged();
            }
        );
    }

    void setController(Controller* controller) {
        this->controller = controller;
    }

    // -----
    // Model notification
    // -----
    void onModelChanged() {
        displayedName = model->getName();
        displayedAge = model->getAge();

        std::cout << "\n[View] Model changed -> View updated\n";

        show();
    }
    // -----
    // Display
    // -----
    void show() const {
        std::cout << "-----\n";
        std::cout << "Name: " << displayedName << '\n';
        std::cout << "Age : " << displayedAge << '\n';
        std::cout << "-----\n";
    }
    // -----
    // User input
    // -----
    void userChangesName(const std::string& newName);

    void userChangesAge(int newAge);
};

```



```

// =====
// Controller
// =====
class Controller {
private:
    Model* model;
    View* view;

public:
    Controller(Model* model, View* view)
        : model(model),
          view(view) {}

    void changeName(const std::string& newName) {
        std::cout << "\n[Controller] Changing name...\n";

        model->setName(newName);
    }

    void changeAge(int newAge) {
        std::cout << "\n[Controller] Changing age...\n";

        model->setAge(newAge);
    }
};
// =====
// View -> Controller
// =====
void View::userChangesName(const std::string& newName) {
    std::cout << "\n[View] User entered name: "
              << newName << '\n';

    controller->changeName(newName);
}

void View::userChangesAge(int newAge) {
    std::cout << "\n[View] User entered age: "
              << newAge << '\n';

    controller->changeAge(newAge);
}
// =====
// Main
// =====
int main() {
    // 1. Model
    Model model("Ali", 20);
    // 2. View
    View view(&model);
    // 3. Controller
    Controller controller(&model, &view);
    // 4. Connect Controller to View
    view.setController(&controller);
    // =====
    // Initial View
    // =====
    std::cout << "Initial View:\n";
    view.show();
}

```

```
// =====
// User changes name
// =====
view.userChangesName("Reza");
// =====
// User changes age
// =====
view.userChangesAge(25);
// =====
// Same value -> no notification
// =====
view.userChangesAge(25);

return 0;
}
```

💡 پاسخ به برخی سوالات و اشتباهات رایج و مهم در MVC:

- 1- گرفتن داده از دیتابیس و ثبت داده در دیتابیس وظیفه لایه Controller است! لایه Model به صورت Runtime در رم، اطلاعات مورد نیاز را نگه می دارد و آخرین نسخه به روز شده مقادیر متغیرها در Model ذخیره می شود و از Model بازیابی می شود! شاید برخی گمان کنند این کار هم به عهده Model است چون مستقیماً با داده سر و کار دارد!
- 2- مثال قابل لمس از معماری MVC در سیستمی مثل ساعت به این شکل است که: View ساعت می تواند دیجیتال یا عقربه دار باشد و Controller همان بخش موتور ساعت است که تیک تاک می کند و Model هم مقدار مربوط به ثانیه و دقیقه و ساعت را در خود نگه می دارد! پس زمانی که Controller یک واحد، عقربه ثانیه شمار یا دقیقه شمار یا ساعت شمار را به جلو هدایت می کند، به Model مقدار جدید را تنظیم می کند و تغییرات واقع در Model به سمت View به اصطلاح Notify می شود و View هم تغییرات جدید را به صورت مکانیکی در ظاهر خود تغییر می دهد! اگر کاربر بخواهد عقربه های ساعت را تنظیم کند، تنظیمات وی از View بدون هیچ تغییری وارد Controller می شود! Controller پس از بررسی صحت تنظیمات، تنظیمات جدید را به Model اطلاع می دهد و در Model تنظیم می کند و تغییرات جدید وارد View شده و View خود را تنظیم و Refresh می کند! نکته زیبای این سیستم (زیرسیستم – Subsystem) آنست که اگر بخواهید ساعتتان را با نمایش Digital به نمایش در آورید یا آنرا عقربه دار کنید و یا اگر بخواهید ظاهر (رنگ و تم) برنامه را عوض کنید، کافیه فقط لایه View را عوض کنید و دیگر با Model و Controller که پیاده سازی منطق کار است سر و کار نخواهید داشت!
- 3- اگر در سیستمی مانند سیستم ثبت سفارش فروشگاه، فرم ثبت سفارش را به چند فرم مختلف تقسیم کنید تا به طور مثال در یکی، گروه بندی سفارش را از کاربر دریافت کنید و در دیگری تعداد سفارش و در دیگری بسته بندی سفارش را، آیا باید برای هر صفحه یک Model و View و Controller جداگانه تنظیم کنید؟ پاسخ خیر است! همه این صفحات یک منطق دارند و آن ثبت یک سفارش در چند بعد است! لذا در این نمونه، شما چند لایه View جداگانه با یک Controller و یک Model مختص ثبت سفارش دارید و لذا Controller به همه این View ها دسترسی دارد و هر View متد مورد نیاز خود از Controller مد نظر را Call می کنند و تغییرات اخیر را از Model دریافت می کنند و ظاهر خودشان را به روز می کنند! پس گاهی پیش می آید که چند View، یک Controller و یک Model مشترک دارند! در این سناریو، Controller به طور کامل شامل تمام Method های مورد نیاز خود است و Model هم به طور کامل شامل تمام Property ها و Setter و Getter های یک Order کامل است! به صورت کلی هر Controller به همراه Model اش، بازیگر یکی از نقش های اصلی نرم افزارتان است، مثلاً بخش ارسال داده های UDP، بخش ارسال و دریافت داده های TCP، بخش محاسبه امتیازات، بخش ثبت سفارش، بخش سرچ سفارشات، بخش نمایش کتگوری های فروشگاه، بخش مدیریت چت یا کنفرانس نرم افزار و ... پس لزوماً هر فرم به

صورت جداگانه، Model و View و Controller اختصاصی ندارد و گاهی چند فرم UI پیوسته و هم هدف مثل فرم های Wizard ثبت سفارش می توانند یک Controller و Model مختص آن را داشته باشند!

4- اگر Controller در کارش نیاز به سرویس (بخوانید Controller) دیگری همانند سرویس ارسال ایمیل یا سرویس ارسال ریکوئست به سرور داشته باشد، می تواند Controller مختص آنرا توسط Constructor دریافت کند و در یک private property نگه دارد و با آن سر و کار داشته باشد! اینجا معمولا Controller دیگر را در Model نگه نمی دارد! Model مختص داده های Primitive مختص Logic اصلی کار است!

5- اگر Model برای کارش نیاز به نگه داری لیستی از Model ساده تری داشته باشد، می تواند آرایه ای از Model های آیتم مورد نیاز را نگه دارد! پس مشاهده می کنید که هر لایه می تواند نوع داده هم نوع خود را در خود نگه دارد! این قانون شامل حال View هم می شود! شاید شما نیاز داشته باشید View بزرگتری را به View های کوچکتری تقسیم کنید و همه این بخش ها را در View اصلی جمع آوری کنید و View اصلی کار اتصال با Controller جهت ارسال دستورات کاربر به آن و اتصال به Model اصلی جهت Refresh شدن خود و دیگر View های داخل خودش از روی نوتیفیکیشن Model را انجام دهد! حتی پیش می آید که شما یک View کلی داشته باشید (مانند صفحه اصلی برنامه) که شامل چندین View مختلف باشد که هر کدام از این View ها، Controller و Model مختص به خود را داشته باشند! برای مثال در عین حالی که بخش Chat را مشاهده می کنید، با زدن دکمه دیگری، View بخش Conference هم به نمایش در آید! پس همانطوری که بخش Chat تحت مدیریت Controller بخش خودش است، View بخش Conference هم تحت مدیریت Controller بخش مختص خودش است و از قضا هر دوی این صفحات در صفحه اصلی قابل مشاهده توأمان با هم هستند!

6- سعی کنید بر اساس قانون اول Solid یعنی Single Responsibility Principle، Controller های برنامه تان را به انواع مختلف بشکنید! لذا قرار نیست تمام Controller هایتان، متخصص اتصال به دیتابیس یا متخصص ارسال مدیریت شده Http Request به سرور یا متخصص برقراری اتصال Http Socket یا ... باشند! برای هر کدام از این موارد، یک Controller به همراه Model مختص آنرا توسعه می دهید و هر Controller ای که نیاز به این کارها داشت، آن Controller را به واسطه Constructor دریافت می کند و کار مدنظر را به واسطه آن Controller انجام می دهد! لذا در این بخش هم ملاحظه می فرمایید که قانون MVC شما را ملزم نمی کند همیشه همه جا Model و View و Controller داشته باشید! گاهی برای برخی عملیات نیازی به View ندارید و فقط Model و Controller کفایت می کند همچنان که ارسال و دریافت اطلاعات به دیتابیس و سرور نیازمند View نیستند!

7- اینک بهترین جایگزین معماری قدیمی N-Tire در نوع Presentation (اپلیکیشن های سمت کلاینت)، معماری های MVC و MVP و MVVM است و در لایه سمت Server هم Clean و Onion و Hexagonal (Port & Adapter) است!

○ معماری MVP: این الگوی معماری، جزو الگوهای معماری لایه پرزنتیشن (Presentation Architectural

pattern) است! این معماری شبیه به معماری MVC است ولی جدیدتر از آن که مخفف Model- View-Presenter است!

در این معماری Presenter یا Mediator نقش Controller را در MVC بازی می کند و مهمترین نکته ای که در این معماری وجود دارد، آن است که یک View تقریباً نقش یک View را با قدرت کمتر به نسبت View در MVC بازی می کند! در واقع View دارای توابع Setter و Getter و برخی توابع دیگر است که توسط Presenter صدا زده می شوند و مقادیر را Set می کند!

مثال: 🧩

کلاس Model: 🎮

```
class UserModel {
public:
    std::string getName() { return "John Doe"; }
};
```

کلاس View: 🖥️

```
class IUserView { // Interface for abstraction
public:
    virtual void showUserName(const std::string& name) = 0;
};

class ConsoleUserView : public IUserView {
public:
    void showUserName(const std::string& name) override {
        std::cout << "User: " << name << std::endl;
    }
};
```

کلاس Presenter یا Mediator: 🧠

```
class UserPresenter {
private:
    UserModel model;
    IUserView* view;

public:
    UserPresenter(IUserView* view) : view(view) {}

    void updateView() {
        std::string name = model.getName();
        view->showUserName(name);
    }
};
```

برنامه اصلی: 🔑

```
int main() {
    ConsoleUserView view;
    UserPresenter presenter(&view);
    presenter.updateView(); // Output: "User: John Doe"
    return 0;
}
```

در این مثال دیدیم بر خلاف MVC که دستورات کاربر به واسطه View به Controller داده می شود، دستورات اصلی توسط Presenter انجام می شوند و View فقط و فقط نقش نمایش دهنده را بازی می کند! از نظر سادگی تست، دیگر برای تست کردن نیازی نیست UI دخیل شود، بلکه

به سادگی می توانید دستورالعمل ها را با Call کردن متد های Presenter انجام دهید! برای توضیح بیشتر، در MVC وقتی View تان شامل یک Button است که کاری را انجام می دهد، در تست کردنش یا مجبورید Button را دستی کلیک کنید (تست نویسی سخت) و یا مجبورید تابعی سمت View را که Click دکمه را به آن متصل کرده اید را به گونه ای Call کنید ولی در اینجا دیگر سنت شکنی نمی کنید و به سادگی تابعی را که Presenter، کار خاصی را روی View یا Model انجام می دهد را Call می کنید! نه نیازی به کلیک فیزیکی دارید و نه نیازی به Call کردن اسلاتی در سمت View و ...! کافیت یک تابع تست بنویسید و در آن مستقیماً به Presenter بگویید تابع خاصی را Call کند تا به نحو احسن آن کار انجام شود! از طرف دیگری بر خلاف MVC که Model مسئول نوتیفیکیشن تغییرات مقادیر بود تا View بر اساس Notification، مقادیر نمایشی خود را به روز کند، این وظیفه نیز بر عهده Presenter یا Mediator است! از نکات بسیار مثبت این الگوی معماری آنست که هم محاسبات، هم تنظیم مقادیر Model و هم به روز کردن View، همه و همه بر عهده Presenter است و وقتی به سادگی متد خاص آنرا Call کنید، هم Model و هم View و هم Presenter به صورت هماهنگ آن سناریو رو به واسطه Presenter انجام می شود و لذا چون مدیریت همه این بخش ها در Presenter متمرکز می شود، تست بسیار ساده تر می شود!

 **نکته مهم:** در مورد فرامین کاربر که با کلیک روی دکمه و .. صدا زده می شوند، ما در MVP از Event ها استفاده می کنیم. به عنوان مثال:

```
// CalculatorModel.h
#pragma once

class CalculatorModel {
public:
    int add(int a, int b) const {
        return a + b;
    }
    int subtract(int a, int b) const {
        return a - b;
    }
};

// ICalculatorView.h
#pragma once
class ICalculatorView {
public:
    virtual ~ICalculatorView() = default;

    virtual int getFirstNumber() const = 0;
    virtual int getSecondNumber() const = 0;

    virtual void showResult(int value) = 0;
};

// CalculatorPresenter.h
#pragma once

#include "CalculatorModel.h"
#include "ICalculatorView.h"

class CalculatorPresenter {
private:
    ICalculatorView& view;
    CalculatorModel model;

public:
    explicit CalculatorPresenter(ICalculatorView& view);

    void onAddClicked();
    void onSubtractClicked();
};
```

```

// CalculatorPresenter.cpp
#include "CalculatorPresenter.h"

CalculatorPresenter::CalculatorPresenter(ICalculatorView& view)
    : view(view) {}

void CalculatorPresenter::onAddClicked() {
    int a = view.getFirstNumber();
    int b = view.getSecondNumber();

    int result = model.add(a, b);

    view.showResult(result);
}

void CalculatorPresenter::onSubtractClicked() {
    int a = view.getFirstNumber();
    int b = view.getSecondNumber();

    int result = model.subtract(a, b);

    view.showResult(result);
}

// ConsoleView.h
#pragma once
#include "ICalculatorView.h"
class CalculatorPresenter;
class ConsoleView : public ICalculatorView {
private:
    int firstNumber{};
    int secondNumber{};
    CalculatorPresenter* presenter{nullptr};
public:
    void setPresenter(CalculatorPresenter& presenter);
    void run();
    int getFirstNumber() const override;
    int getSecondNumber() const override;
    void showResult(int value) override;
};

// ConsoleView.cpp
#include "ConsoleView.h"
#include "CalculatorPresenter.h"
#include <iostream>
void ConsoleView::setPresenter(CalculatorPresenter& presenter) {
    this->presenter = &presenter;
}

void ConsoleView::run() {
    if (presenter == nullptr) {
        std::cout << "Error: Presenter is not initialized.\n";
        return;
    }

    std::cout << "Enter first number: ";
    std::cin >> firstNumber;

    std::cout << "Enter second number: ";
    std::cin >> secondNumber;

    std::cout << "\n";
    std::cout << "1) Add\n";
    std::cout << "2) Subtract\n";
}

```

```

std::cout << "Choose an operation: ";

int choice;
std::cin >> choice;

switch (choice) {
case 1:
    presenter->onAddClicked();
    break;

case 2:
    presenter->onSubtractClicked();
    break;

default:
    std::cout << "Invalid choice.\n";
    break;
}
}

int ConsoleView::getFirstNumber() const {
    return firstNumber;
}

int ConsoleView::getSecondNumber() const {
    return secondNumber;
}

void ConsoleView::showResult(int value) {
    std::cout << "Result: " << value << '\n';
}

// main.cpp
#include "CalculatorPresenter.h"
#include "ConsoleView.h"
int main() {
    ConsoleView view;

    CalculatorPresenter presenter(view);

    view.setPresenter(presenter);

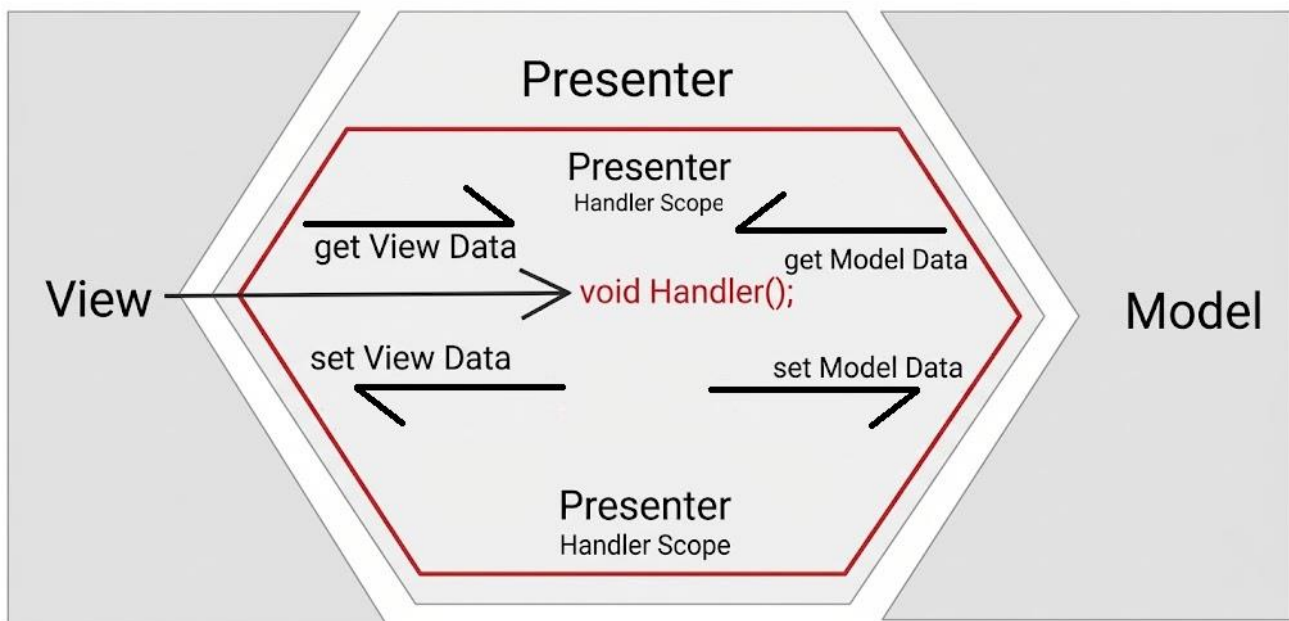
    view.run();

    return 0;
}

```

پس تمام فرامین از View به سمت Callback های با پیشوند On در Presenter می رود تا Presenter با Get کردن داده های مورد نیازش از Model یا View، محاسبات لازم را انجام دهد و اقلام مورد نیازش را در View به روز نماید! همچنان که در مثال دیدید، این بار نقش Presenter بسیار پررنگ از Controller در MVC است و همچنین نقش View شما با داشتن مسئولیت Get و Set بسیار منفعل تر از View در MVC است و همین عامل باعث سادگی تست می شود! در واقع لازم نیست حتما دکمه ای کلیک شود تا تست عملکرد دکمه محور شما شروع شود! بلکه کافیه همان متدهای با پیشوند On در Presenter توسط تابع تست شما ران شود تا تست شما شروع شود و هم روی View و هم روی Model تاثیرات تست را مشاهده کنید! به صورت کلی، Handler های Presenter شامل توابع void بدون پارامتر ورودی با پیشوند نام On هستند که با هر فشار دکمه و ...، Call می شوند و خود این Handler ها از View یا Model مقادیر مورد نیازشان را Get یا Set می کنند محاسباتشان را انجام می دهند و رویه نرم افزار را جلو می برند! با این اختیار بیشتر در MVP، تست شما ساده می شود چون برای سناریوی تست همان Handler، همان را Call کرده و نتیجه را در Model و View و Presenter مشاهده می کنید!

در MVC، ارسال Notification مربوط به تغییر در مقادیر Model. کار Model است و نتیجه آن مستقیماً وارد View می شود و آنرا فوراً به روز می کند! در MVP این وظیفه در دستان Presenter است و پس از تغییرات Model، خودش مقادیر View را Set و به روز می کند و دستور Update یا Refresh آنرا هم می تواند Call کند! همچنین در MVC وقتی دکمه ای Call می شود یا به هر دلیلی، درخواست کاربر به Presenter ارسال می شود، View یکی از توابع Controller را با پاس دادن مقادیر مورد نیازش به همان تابع، صدا می زند و Controller پس از دریافت داده ها از آرگمان ورودی تابع، تصمیمات را می گیرد و محاسبات را انجام می دهد و تغییرات را در Model اعمال می کند و تغییرات Model توسط خود Model به صورت Notification ارسال می شود و این اعلان به دست View می رسد و View خود را به روز می کند! اما در MVP، با ارسال درخواست کاربر توسط View، تابع Handler بدون ورودی و بدون خروجی مختص آن درخواست در Presenter توسط View صدا زده می شود و این Handler، مقادیر مورد نیازش از View را دریافت کرده (Get) و مقادیر مورد نیاز در Model را به روز می کند (Set) و تغییرات مورد نیاز View را اعمال می کند (Set) و اگر لازم شد، View را با صدا زدن تابع Refresh یا Update به روز می کند! شکل مفهومی زیر به نوعی برای درک بهتر ساختار MVP طراحی شده است! شش ضلعی وسط به نوعی به هر دو طرف اشراف کامل دارد و View با صدا زدن Handler مورد نیازش که رنگ قرمز دارد، باعث می شود Handler هر مقداری را که لازم دارد را از Model و View دریافت یا تنظیم کند و محاسباتش را انجام دهد! می توان گفت نقش Model (در عدم ارسال Notification) و نقش View (در کاهش اختیارات در حد فقط Call کردن یک Handler بدون ورودی و خروجی در سمت Presenter) در MVP کم رنگ تر شده و در عوض هر دوی این نقش ها به دوش Presenter سپرده شده است!



این معماری مخفف (Model-View-ViewModel) است! در این معماری لایه مدل، به لایه های دیگر دسترسی ندارد و Property های اولیه یا Primitive را هندل می کند و با استفاده از متدهای مناسب Getter یا Setter یا Handler آنها را اعتبار سنجی و مدیریت می کند!

● لایه Model:

```
// Model.h
#pragma once
class CounterModel {
public:
    CounterModel() : count(0) {}

    void increment() { count++; }

    void decrement() { count--; }

    void reset() { count = 0; }

    int getCount() const { return count; }
private:
    int count;
};
```

● لایه **ViewModel**: این لایه فقط به لایه مدل دسترسی دارد و شیی از مدل را در خود نگه می دارد! همچنین شی یا اشیایی از **CallBack** در خود نگه می دارد! توسط **Handler** های خودش که متقارن با **Handler** های داخل **Model** است می تواند مدل را به شرط صلاحدید مودیفای کند و یا کارهای جانبی لازم را انجام دهد! با هر تغییر در مدل هم می تواند توابع **CallBack** را به شرطی که **null** نباشند به شکل نوتیفیکیشن **Call** کند!

```
// ViewModel.h
#pragma once
#include "Model.h"
#include <functional>

class CounterViewModel {
public:
    CounterViewModel(CounterModel& model) : model(model) {}
    // Commands exposed to View
    void increment() {
        model.increment();
        notifyCountChanged();
    }
    void decrement() {
        model.decrement();
        notifyCountChanged();
    }
    void reset() {
        model.reset();
        notifyCountChanged();
    }
};
```

```

// Data exposed to View
int getCount() const {
    return model.getCount();
}
// set Simple notification callback function
void setCountChangedCallback(std::function<void()> callback) {
    countChangedCallback = callback;
}
private:
    CounterModel& model;
    std::function<void()> countChangedCallback; // function pointer as callback

    void notifyCountChanged() {
        if (countChangedCallback) {
            countChangedCallback();
        }
    }
};

```

🧠 **لایه View:** این لایه همان لایه پرزنتیشن است که شامل دسترسی مستقیم به لایه **ViewModel** است! پس با دستورات کاربر، هندلرهای لایه **ModelView** را کال می کند و تابع **UpdateDisplay** خود را به عنوان **CallBack** نوتیفیکیشن به **ViewModel** می دهد و با این کار، به دستور کاربر یا هر عامل دیگری، **View** هندلرهای مناسب لایه **ModelView** را **Call** می کند و اگر تغییری در مدل توسط **ModelView** رخ دهد، تابع **CallBack** نوتیفیکیشن را **Call** می کند تا بدین وسیله، **View** هم به روز شود!

```

// View.h
#pragma once
#include "ViewModel.h"
#include <iostream>
class CounterView {
public:
    CounterView(CounterViewModel& viewModel) : viewModel(viewModel) {
        // Set up notification callback
        viewModel.setCountChangedCallback([this]() { updateDisplay(); });
    }
    void display() {
        updateDisplay();
        while (true) {
            std::cout << "\nOptions: 1) Increment 2) Decrement 3) Reset 4) Exit\n";
            int choice;
            std::cin >> choice;
            switch (choice) {
                case 1: viewModel.increment(); break;
                case 2: viewModel.decrement(); break;
                case 3: viewModel.reset(); break;
                case 4: return;
                default: std::cout << "Invalid choice!\n";
            }
        }
    }
private:
    CounterViewModel& viewModel;

    void updateDisplay() {
        std::cout << "\nCurrent Count: " << viewModel.getCount() << "\n";
    }
};

```

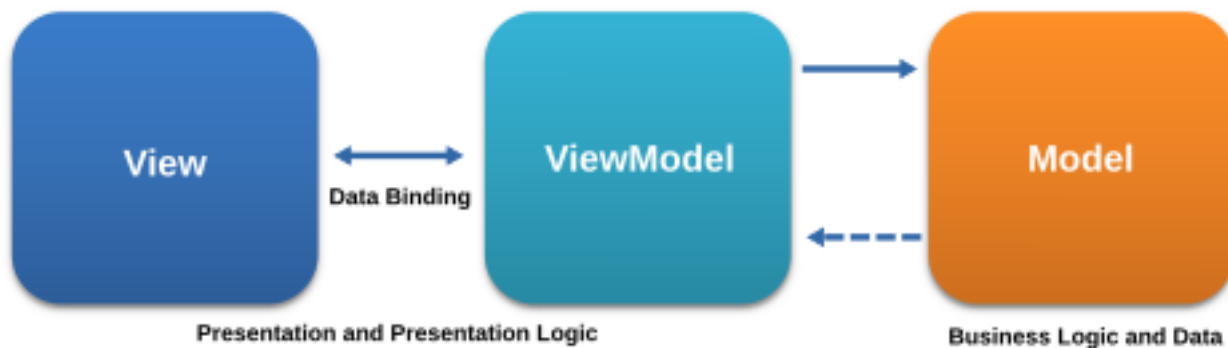
```
// main.cpp
#include "Model.h"
#include "View.h"
#include "ViewModel.h"

int main() {
    // Create and connect components
    CounterModel model;
    CounterViewModel viewModel(model);
    CounterView view(viewModel);
    // Start the UI
    view.display();

    return 0;
}
```

پس در معماری MVVM لایه View فرامین را مستقیماً از کاربر یا اکتور خاصی دریافت می‌کند و خودش شامل لایه ViewModel است و به آن دسترسی مستقیم دارد ولی به صورت مستقیم به لایه Model دسترسی ندارد و Callback (های) خاصی را هم به ViewModel می‌دهد تا در صورت اعلان، آن Callback (ها) توسط ViewModel به نوعی Call شوند. لایه ViewModel هم به View دسترسی ندارد ولی به Model دسترسی مستقیم دارد و شامل آن است! پس Callback (هایی) را از View دریافت می‌کند تا در صورت تغییرات خاصی روی مدل آن (ها) را Call کند! این لایه به واسطه Getter و Setter و Handler های لایه مدل، در صورت صلاحدید، درخواست های خود را به Model می‌دهد.

لایه Model هم شامل Property های نوع primitive به صورت private است که می‌تواند آنها را توسط این Setter و Getter و Handler ها، مودیفای و مدیریت نماید.



📌 **نتیجه گیری در انتخاب بین معماری های لایه پرزنتیشن:** برای UI های ساده از MVC و برای فریم ورک ها و UI های پیچیده از MVVM استفاده کنید و برای تست پذیری بهتر و جدا کردن منطق برنامه از نمایش و داده می‌توانید از MVP استفاده کنید تا Presenter دستور دهنده اصلی کار باشد! معمولاً در پروژه های مختلف جهان، انتخاب اول و معروف MVC است ولی با مواردی که گفته شد، به نظر می‌رسد MVP انتخاب بهتر و هوشمندانه تری باشد!

○ معماری (DDD) Domain Driven Design: این معماری، خودش یک Architectural Pattern نیست!

بلکه یک رویکرد/روش طراحی نرم‌افزار است که اصول و الگوهایی برای طراحی سیستم حول Domain ارائه می‌دهد، ولی چون پایه و زیرساخت 3 مدل معروف معماری نرم‌افزار Backend است، توضیح آن در این بخش خالی از لطف نیست!



Domain-Driven Design یا طراحی مبتنی بر دامنه یا طراحی دامنه محور یا به صورت خلاصه DDD، رویکردی برای توسعه نرم‌افزار است که اریک ایوانز (Eric Evans) در سال ۲۰۰۳ در کتاب معروف خود یعنی:

Domain-Driven Design: Tackling Complexity in the Heart of Software

آن را معرفی کرد. ایده اصلی DDD ساده اما عمیق است: پیچیدگی واقعی نرم‌افزارهای سازمانی از دامنه کسب‌وکار (Business Domain) می‌آید، نه از فناوری! بنابراین باید تمرکز اصلی تیم توسعه روی درک عمیق دامنه، مدل‌سازی صحیح آن، و همسو نگه‌داشتن کد با این مدل باشد.

DDD صرفاً یک الگوی معماری نیست؛ ترکیبی از یک فلسفه طراحی، مجموعه‌ای از الگوهای استراتژیک برای سازمان‌دهی سیستم‌های بزرگ و مجموعه‌ای از الگوهای تاکتیکی برای پیاده‌سازی کد در سطح دامنه است. در اصل DDD یک رویکرد در توسعه نرم‌افزار است که تمرکز روی درک و مدل‌سازی صحیح دامنه کسب و کار را بر عهده دارد. در این رویکرد، مفاهیم کسب و کار به عنوان مدل‌های دامنه مدل‌سازی می‌شوند و تمامی اعضای تیم می‌توانند به درک مشترکی از دامنه مسئله برسند و با سایر اعضای تیم، بهبود یافته و کارآمدتر همکاری کنند. از مهمترین مسایلی که در DDD مورد توجه قرار می‌گیرد، چالش‌های اصلی یک بیزنس است و تلاش اصلی تیم حول مسایل اصلی آن بیزنس متمرکز می‌شود! برای مثال در مورد کسب و کار دیجیکالا که Domain یا دامنه آن، خرده فروشی آنلاین است، درگیری بخش Search & Filter برای یافتن یک محصول، 10 برابر شلوغ تر و پرکارتر از بخش خرید محصول است! بنابراین برای کسب سرعت و کیفیت بالاتر، بهتر است زیست این دو بخش به صورت جدا در 2 سرور یا Container مجزا رقم بخورد تا با از کار افتادن یکی، دیگری از کار نیفتند و از طرفی درگیر شدن بیش از اندازه یک قسمت، روی کار بعدی

تاثیر نگذار! البته این بحث به سیستم های توزیع شده یا به اصطلاح Microservice ها مربوط می شود ولی به نوعی می توان گفت: منابع و مراجع و علم طراحی سیستم های Microservice به معماری DDD گره خورده است!

DDD به چه کاری می آید؟ معمولاً در پروژه های نرم افزاری بزرگ، با چند مشکل تکرارشونده مواجه می شویم:

1- **شکاف بین تحلیل گران کسب و کار و برنامه نویسان:** تیم کسب و کار با زبانی صحبت می کند که با زبان تیم فنی متفاوت است. این باعث سوء تفاهم، مستندات نادرست و در نهایت نرم افزاری می شود که نیاز واقعی را برآورده نمی کند!

2- **مدل های داده محور به جای رفتار محور:** بسیاری از سیستم ها صرفاً حول جداول پایگاه داده طراحی می شوند (الگوی Anemic Domain Model) که منطق کسب و کار در آن ها پراکنده و در لایه سرویس ها تکرار می شود.

3- **رشد بی رویه کد و از دست رفتن مرزهای منطقی:** با بزرگ شدن پروژه، بدون مرزبندی مشخص، همه چیز به هم وابسته می شود و تغییر در یک بخش، بخش های دیگر را به چالش می کشد و عملکرد آنها را هم تحت شعاع قرار می دهد! (توپ بزرگ گلی - Big Ball of Mud)!

4- **دشواری در مقیاس پذیری تیم:** وقتی چند تیم روی یک Code Base بزرگ کار می کنند، بدون مرزهای واضح، تداخل و کندی توسعه اجتناب ناپذیر است.

DDD با ارائه ابزارهایی مانند زبان مشترک (Ubiquitous Language) و مرزهای دامنه (Bounded Context) این مشکلات را هدف قرار می دهد.

مهم ترین درس DDD این است که کد باید بازتابی صادق از درک تیم از کسب و کار باشد، نه انعکاسی از ساختار پایگاه داده یا الگوهای فنی محبوب. DDD ابزاری قدرتمند است، اما مثل هر ابزاری باید متناسب با پیچیدگی واقعی مسئله به کار رود؛ نه هر جا.

🏠 معماری لایه های DDD: DDD معمولاً در قالب یک معماری لایه ای (یا Hexagonal یا Clean) پیاده سازی می شود تا دامنه از جزئیات فنی جدا بماند. جداسازی دامنه از فناوری، معمولاً هم راستا با Clean Architecture یا Hexagonal Architecture یا Onion Architecture رقم می خورد!

عملیات طراحی مدل بیزنس بر پایه DDD به دو دسته کلی تقسیم می شود:

بخش	تمرکز
Strategic Design	مدیریت پیچیدگی در سطح کلان سازمان و سیستم
Tactical Design	الگوهای پیاده سازی در سطح کد

🏠 طراحی استراتژیک (Strategic Design): **مهم ترین بخش طراحی DDD** طراحی استراتژیک به سؤال «چگونه سیستم

بزرگ را به بخش های قابل مدیریت تقسیم کنیم؟» پاسخ می دهد! به بیان بهتر یعنی تقسیم سیستم به Bounded Context های مستقل با زبان مشترک مخصوص کارکنانشان. به بیانی دیگر یعنی جدا کردن بخش های مختلف و مرزبندی بخش های مهم یک پروژه!

در طراحی استراتژیک روی Core Domain بیشترین سرمایه گذاری را انجام می دهیم ولی برای Generic Subdomain می توانیم از راه حل های آماده استفاده کنیم. (در قسمت های بعد با این اصطلاحات آشنا خواهید شد!)

 **زبان مشترک (Ubiquitous Language):** زبان مشترک، واژگانی است که بین متخصصان دامنه مدنظر (Domain Experts) و تیم توسعه به طور یکسان استفاده می‌شود و مستقیماً در کد نیز منعکس می‌شود. اگر متخصص کسب و کار، از واژه «سفارش تأییدشده» استفاده می‌کند، کد نیز باید دقیقاً از همین اصطلاح (مثلاً کلاس `ConfirmedOrder` یا متد `Confirm`) استفاده کند، نه اصطلاحات فنی بی‌ربط مانند:

Status = 2

زبان مشترک بین توسعه‌دهندگان و متخصصان دامنه (Domain Experts) ویژگی‌های زیر را دارد:

- در کد، مستندات، و گفتگوها یکسان استفاده می‌شود!
- از ابهام و ترجمه‌های نادرست جلوگیری می‌کند!
- این زبان مشترک در کل سازمان نیست؛ بلکه مختص هر `Bounded Context` است (همان کلمه در دو زمینه متفاوت می‌تواند معنای متفاوتی داشته باشد).

 **مثال:** اگر متخصص کسب و کار می‌گوید "Customer places an Order"، کد باید دقیقاً همین اصطلاحات را داشته باشد:

```
customer.placeOrder(orderDetails);
```

نه چیزی مثل موارد زیر:

```
customer.createOrderRecord() یا customer.insertOrderData()
```

بخش استراتژیک تلاش می‌کند با روش‌های مختلف از جمله زبان اصناف (اینکه ادبیات یک صنف یا بخشی مثل حسابداری چیست و تفاوتش با ادبیات بخش‌های دیگر مثل انبارداری چیست)، این بخش‌ها را از هم جدا کند و بخش‌های جدا شده را به دست بخش تکنیکال بدهد تا با پترن‌های مناسب آنرا پیاده‌سازی کند!

مهم‌ترین درس `DDD` در این بخش این است که کد باید بازتابی صادق از درک تیم از کسب و کار باشد نه انعکاسی از ساختار پایگاه داده یا الگوهای فنی محبوب. `DDD` ابزاری قدرتمند است، اما مثل هر ابزاری باید متناسب با پیچیدگی واقعی مسئله به کار رود؛ نه هر جا.

 **طراحی تاکتیکی:** مدل‌سازی و برنامه‌نویسی داخل هر `Context` با استفاده از `Domain`, `Aggregate`, `Value Object`, `Entity`

`Factory` و `Domain Service`, `Repository`, `Event` انجام می‌شود. بخش تکنیکال یا بلاک ساخت مدل، مجموعه‌ای از پترن‌هایی است که بوجود آوردن مدل‌های موثر برای `Context`‌های پیچیده مرتبط را ممکن می‌کند. در این مرحله، تصمیم می‌گیریم بخش خاصی که استراتژی آنرا جدا کرده ایم را چگونه و با چه زبان و `Design Pattern`‌هایی طراحی کنیم!

 برای اینکه دو مرحله `Strategic` و `Tactical` را پیاده کنید، باید مراحل زیر را به صورت تیمی در کنار `Domain Expert`‌ها و تیم توسعه پیش ببرید! (اصطلاحات ناآشنای مطرح شده در گزینه‌های زیر را در بخش‌های بعدی این کتاب توضیح داده ایم!)

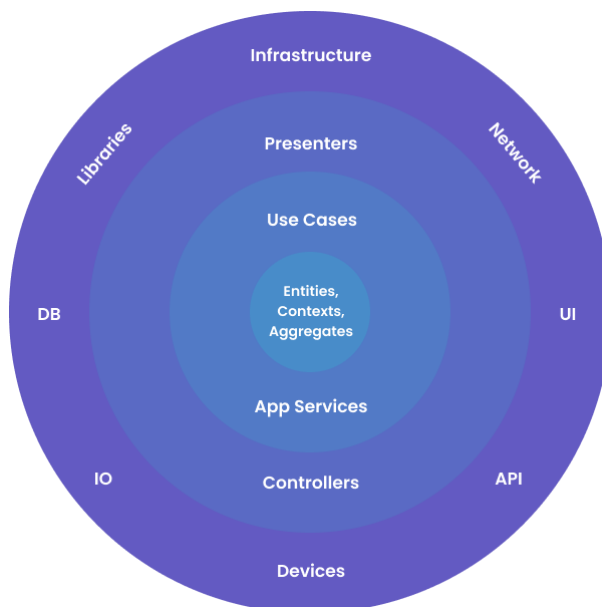
1. **Event Storming:** یک تکنیک `Workshop` محور برای کشف `Domain Events` با مشارکت متخصصان کسب و کار. در یک `Workshop` برای کشف `Domain` ها و `Sub Domain` ها می‌توانید از `Event Storming` به روش `Alberto Brandolini` استفاده کنید! (نیاز به مطالعه در وب یا کلاسهای استاد آزاد)

2. **شناسایی Bounded Contexts:** بر اساس `Event Storming`، مرزهای منطقی مشخص می‌شود.

3. **مدل‌سازی Aggregate:** شناسایی `Entity` ها، `Value Object` ها و مرزهای `Aggregate`.

4. پیاده‌سازی لایه **Domain**: برنامه نویسی Rich Domain Model مستقل از Infrastructure.
5. اتصال به **Infrastructure**: پیاده‌سازی Repository، Application Services و ارتباط با معماری کلی.

🔧 لایه های معماری DDD:



🧠 **Domain Layer**: لایه مرکزی شامل Entity ها، Value Object ها، Aggregate ها، Domain Service ها، Domain Event ها و رابط Repository (Interface) ها است. این لایه هیچ وابستگی به فریم‌ورک یا پایگاه داده ندارد و باید کاملاً قابل تست با Unit Test باشد.

🎯 **Application Layer**: مسئول اجرای Use Case ها و Orchestration است! در واقع درخواست های کاربران از لایه سوم یعنی Presentation به این لایه وارد می شود و این لایه با در اختیار داشتن لایه مرکزی یعنی Domain، درخواست کاربران را اجرا می کند!

🖥 **Presentation Layer**: این لایه جهت تعامل با کاربران از انواع مختلف (Actors) به کار می رود! (کنترلرهای REST API، GraphQL Resolver ها، یا رابط کاربری)

🔑 **Infrastructure Layer**: پیاده‌سازی واقعی Repository ها است (مثلاً با Entity Framework یا Dapper)، ارتباط با سرویس‌های خارجی مثل سرویس پرداخت یا سرویس ارسال SMS و Email، پیام‌رسانی (Message Broker)، دیتابیس های مختلف و غیره.

خلاصه روند مکانیسم DDD: کاربر درخواست خود را از طریق لایه سوم، یعنی Presentation Layer، به سیستم ارسال می کند.

در این لایه، Route ها، درخواست‌های کاربر را به متدهای مختلف Controller مربوطه متصل می‌کنند! Controller ها نیز درخواست را به لایه دوم، یعنی Application Layer، منتقل می‌کنند.

در لایه Application، سرویس‌هایی وجود دارند که درخواست کاربر را به صورت Primitive و Parametric دریافت می‌کنند و با در اختیار داشتن Domain Model ها (مدل Entity ها)، امکان اجرای درخواست را دارند.

اما برای اجرای این درخواست‌ها، Application باید به Repository مربوط به Domain موردنظر دسترسی داشته باشد.

اینجا یک سؤال مهم مطرح می‌شود:

چگونه می‌توان Repository موجود در لایه چهارم را در اختیار لایه دوم قرار داد، بدون اینکه لایه دوم به لایه چهارم وابسته شود (include کند)؟
به بیان ساده‌تر، چگونه می‌توان از یک متد در لایه دوم، متدی از یک شیء در لایه چهارم را صدا زد، بدون اینکه کلاس لایه چهارم را در لایه دوم include کنیم؟

💡 پاسخ: **Dependency Injection!** اینجاست که جادوی DI قد علم می‌کند!

در لایه دوم، برای Repository‌های مختلف، Interface تعریف می‌کنیم. برای مثال، برای User که دارای Id و Name است، یک Interface به نام IUserRepository در لایه دوم تعریف می‌کنیم.

```
#include <string>
#include <memory>
//-----Domain (Entity) in Layer 1 (Domain or Entity Folder)
//domain/user.h
class User {
private:
    int id_;
    std::string name_;
public:
    User(int id, std::string name) : id_(id), name_(std::move(name)) {}

    int getId() const { return id_; }
    const std::string& getName() const { return name_; }
};
//-----User Repository interface in Layer 2 (Application layer folder)
application/repository/IUserRepository.h
class IUserRepository {
public:
    virtual ~IUserRepository() = default;
    virtual std::shared_ptr<User> findById(int id) = 0;
    virtual void save(const User& user) = 0;
};
```

حالا لایه دوم هم Domain را (User در لایه 1) در اختیار دارد و هم Interface مربوط به Repository آن را (IUserRepository در لایه 2). سپس با استفاده از Dependency Injection، پیاده‌سازی واقعی Repository را از بیرون به لایه دوم تزریق می‌کنیم.

```
//infrastructure/postgresql/UserRepository.hpp
#include <memory>
// IUserRepository interface is defined in Layer 2
#include "application/repository/IUserRepository.h"
// User entity is defined in Layer 1
#include "domain/User.h"
//----- PostgreSQL Repository implementation in Layer 4 (Infrastructure)
class PostgresUserRepository : public IUserRepository {
public:
    std::shared_ptr<User> findById(int id) override {
        // PostgreSQL implementation goes here.
        // Execute SELECT query and map the result to User.
        return nullptr;
    }
    void save(const User& user) override {
        // PostgreSQL implementation goes here.
        // Execute INSERT/UPDATE query using user.getId() and user.getName().
    }
};
```


در نتیجه، Domain و Application از طریق Interface به Repository متصل می‌شوند، بدون اینکه مستقیماً به پیاده‌سازی دیتابیس وابسته باشند.

برای مثال، UserRepository مربوط به PostgreSQL در لایه چهارم، IUserRepository موجود در لایه دوم را پیاده‌سازی می‌کند و به دیتابیس PostgreSQL متصل می‌شود.

در نهایت، در Composition Root (تابع main) یک نمونه واقعی از PostgresUserRepository می‌سازیم و آن را از طریق Constructor به سرویس موردنظر در Application Layer (که از شما یک IUserRepository دریافت می‌کند) تزریق می‌کنیم. به این ترتیب، درخواست کاربر در لایه دوم اجرا می‌شود و در صورت نیاز، از طریق Repository واقعی به دیتابیس در لایه چهارم دسترسی پیدا می‌کند.

نکته حیاتی: در ساختارهای DDD بیس، جهت وابستگی همیشه از لایه‌های بیرونی به لایه‌های داخلی (به سمت Domain) است، به بیان بهتر لایه Domain مرکزی نباید به هیچ لایه دیگری و نه به پایگاه داده یا فریم ورک Web وابسته باشد! لایه Application فقط به لایه Domain وابسته است و آنرا می‌شناسد (Include می‌کند)! لایه Presentation هم به لایه‌های Domain و Application وابسته است و آنها را می‌شناسد! لایه بیرونی یعنی Infrastructure هم به تمام 3 لایه وابسته است و آنها را می‌شناسد! این اصل همان چیزی است که در Onion Architecture و Clean Architecture و Hexagonal Architecture (Ports & Adapters) نیز محور اصلی است. به بیان بهتر DDD، Onion Architecture، Clean Architecture و Hexagonal Architecture مکمل هم هستند: DDD به مدل‌سازی دامنه می‌پردازد، در حالی که آن سه الگو نحوه سازمان‌دهی فیزیکی لایه‌ها و وابستگی‌ها را مشخص می‌کنند. در ادامه کتاب، به تشریح این الگوهای معماری خواهیم پرداخت!

با درک این مفهوم، بخش مهمی از نحوه کار معماری‌های Onion، Clean Architecture و Hexagonal Architecture نیز قابل درک می‌شود.

دامنه و زیردامنه‌ها (Domain & Subdomains): دامنه (Domain) به‌طور کلی حوزه کسب‌وکاری است که سازمان در آن فعالیت می‌کند (مثلاً تجارت الکترونیک). دامنه معمولاً به چند زیردامنه (Subdomain) تقسیم می‌شود:

Core Domain (دامنه هسته‌ای): بخشی که مزیت رقابتی اصلی کسب و کار در آن نهفته است و به نوعی سازمان شما در آن مورد شهره است. این بخش باید بیشترین دقت، زمان، هزینه و بهترین توسعه‌دهندگان را دریافت کند. مثلاً برای یک شرکت حمل و نقل، الگوریتم مسیریابی هوشمند Core Domain است، یا برای یک شرکت پرداخت یاری، امنیت پرداخت و بخش Core مرکز مدیریت پرداخت Core Domain است! برای یک شرکت مرکزی بورس، هسته معاملات بورسی Core Domain است!

مثالی از اهمیت تمرکز روی Core Domain: قطعاً Core Domain بانک رسالت که مختص وام دهی است با Core Domain بانک تجارت که متخصص ترانکش های بانکی است، متفاوت است و نیازمند تمرکز روی اولویت های مختلفی دارند! معمولاً یک دامنه به چند زیردامنه (Subdomain) تقسیم می‌شود! (Subdomain بخش هایی از سیستم است که نسبت به بخش های دیگر ایزوله هستند و بخشی از یک بخش بزرگتر هستند)

Supporting Subdomain (زیردامنه پشتیبان): بخش‌هایی که برای کسب و کار لازم‌اند اما مزیت رقابتی محسوب نمی‌شوند. به بیان بهتر از Core Domain پشتیبانی می‌کند ولی خودش مزیت رقابتی اصلی نیست! این‌ها معمولاً اختصاصی توسعه داده می‌شوند اما با سرمایه‌گذاری کمتر. برای مثال برای شرکت آمازون یا دیجیکالا، بخش Shipping و Delivery و ارسال بسته‌ها می‌تواند جزو این دسته از Subdomain ها باشد! قطعاً یکی از دلایل انتخاب این شرکتها برای خرید محصولات، Delivery راحت و پیشرفته و حرفه ای آنهاست، ولی

دلیل اصلی نیست! برای بخش Supporting Domain معمولاً پیشنهاد می شود یا سیستم آنرا آماده بخرید یا با دولوپرهای معمولی تر آنرا بنویسید.

Generic Subdomain (زیردامنه عمومی): عملکردهایی که در همه کسب و کارها مشابه و معمولند (مثل احراز هویت، پرداخت، ارسال ایمیل و اس ام اس، ارتباط با مشتریان، آزمون ساز آنلاین جهت نظرسنجی، ارتباط Voip داخلی شرکت). معمولاً بهتر است این ها از راه حل های آماده یا سرویس های ثالث استفاده شوند، نه توسعه از صفر. چرا که تمرکز شرکت های توسعه دهنده آنها، در کنار هزینه کردشان و گرفتن بازخورد مشتریان و به علاوه، تامین هزینه توسعه این سرویس ها توسط هزاران مشتری با نیازهای روتین و مشترک، باعث می شود اختراع دوباره چرخ توسط ما به صرفه نباشد و می توانیم این بخش ها را با هزینه پایین تر به دست بیاوریم و تمرکزمان را روی Core Domain و Supporting Domain بگذاریم!

پس به صورت خلاصه، Generic Subdomain یعنی قابلیت عمومی و استاندارد که معمولاً می توان آن را خرید یا به صورت آماده استفاده کرد.

نکته مهم در مورد هر سه نوع Sub Domain

معمولاً Core Domain تغییر نمی کند مگر با تغییر Domain ها! از طرف دیگری، ما با تغییر Supporting Domain و Generic Domain نیازی به تغییر Core Domain نداریم و یا حداقل تغییر در این بخش ها نمی تواند Core Domain را مجبور به تغییر کند! این مورد درباره Supporting Domain و Generic Domain هم صادق است و Generic Domain نمی تواند با تغییرش، Supporting Domain را تغییر دهد.

مرز دامنه (Bounded Context): (زمینه محدود یا زمینه مرزبندی شده) مهم ترین مفهوم استراتژیک DDD است. این مفهوم مرزی صریح (منطقی و اغلب فنی) تعریف می کند که درون آن یک مدل دامنه خاص و یک زبان مشترک خاص معتبر است. **مثال:** در یک سیستم فروشگاهی، مفهوم «محصول» (Product) در Bounded Context «کاتالوگ» شامل توضیحات، تصاویر و دسته بندی است؛ اما همان «محصول» در Bounded Context «انبار» شامل موجودی، مکان قفسه و وضعیت رزرو است. این دو مدل، با اینکه هر دو «محصول» نامیده می شوند، از نظر داده و رفتار کاملاً متفاوت اند و نباید در یک مدل واحد ادغام شوند!

کلمه **Product** در:

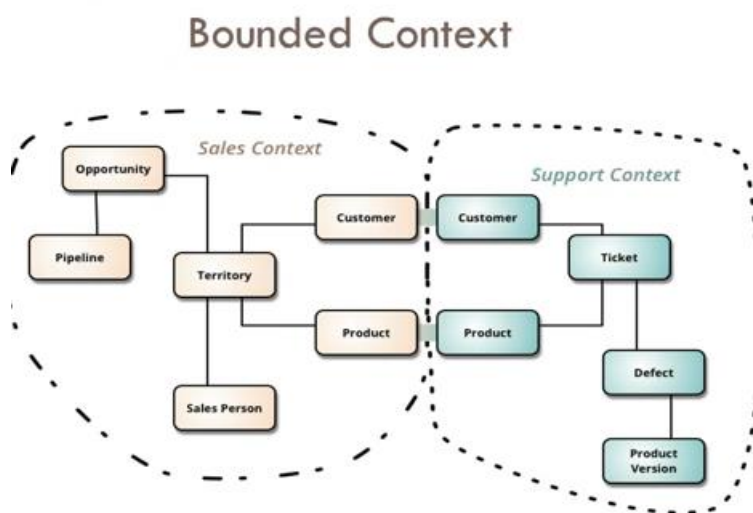
- **Bounded Context فروش:** قیمت، موجودی، تخفیف

- **Bounded Context انبار:** مکان قفسه، وزن، ابعاد

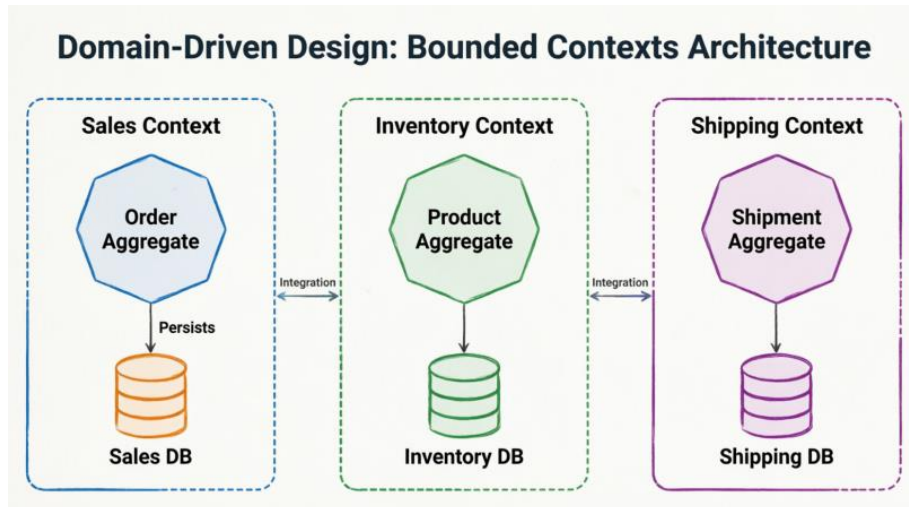
- **Bounded Context حسابداری:** هزینه تمام شده، مالیات

وقتی ما از Bounded Context صحبت می کنیم، منظورمان تقسیم کردن بخش ها و پازل های (زیرسیستم ها) یک مجموعه (سیستم) به بخش های مختلفی است که هر بخش، ادبیان و زبان فراگیر و خاص خود (Ubiquitous Language) را بین اهالی مشغول به آن دارد! برای مثال وقتی وارد مقوله انبار می شوید، اصطلاحات خاصی بین انبارداران رایج است! یا وقتی وارد بخش حساب داری شرکت می شوید، ادبیات خاصی بین حسابداران رایج است! همین طور وقتی وارد بخش HR می شوید، ادبیات خاص خود را دارند! منظور از ادبیات هر بخش، کلمات رایج و پرتکراری است که افراد هر بخش با همکاران بخش خود دارند! همین تقسیم بندی Bounded Context ها بر اساس ادبیات بخش های مختلف خود می تواند یکی از بهترین معیارها برای جداسازی توسعه سیستم های بخش های مختلف به عنوان Bounded Context باشد!

هر Bounded Context معمولاً معادل یک سرویس مستقل (در معماری Microservice) یا حداقل یک ماژول مجزا در یک مونولیت (Modular Monolith) است.



اگر پروژه شما Microservice (توزیع شده) باشد، معمولاً هر Bounded Context، تیم و دیتابیس و منطق خاص خود را دارد و در نتیجه هر Bounded Context یک میکروسرویس است! پس هیچ Bounded Context نباید مستقیم به دیتابیس Bounded Context سرویس دیگری دسترسی و دستور دهی داشته باشد بلکه باید از طریق Aggregate Root مربوط به آن سرویس، درخواست مجاز خود را صادر کند و اگر تغییری قرار است روی Database رخ دهد، توسط Aggregate Root اتفاق بیفتد! در بخش های بعدی این فصل با Aggregate Root آشنا خواهید شد!



هر SubDomain به صورت ساختاری یک Bounded Context است!

هر Bounded Context یک دسته بندی است که باعث جداسازی و تفکیک SubDomain ها می شود.

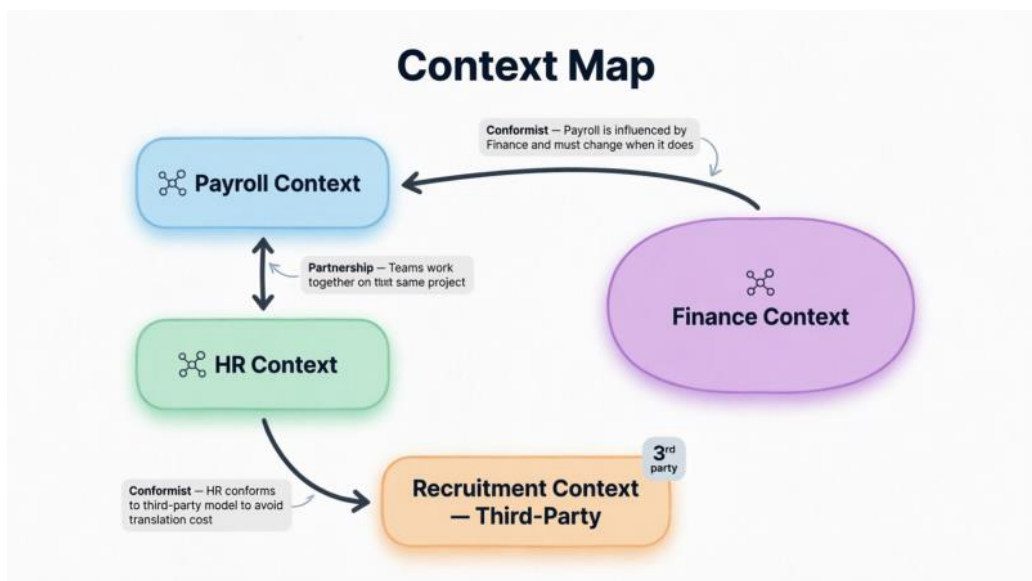
هر Bounded Context زبان خاص خود را دارد. پس تکنولوژی و هدف خاص خود را دارد و یک زیر سیستم است!

پس در نتیجه تا اینجا هر Domain از همکاری چند SubDomain ایجاد می شود و هر SubDomain در فاز تحلیل و پیاده سازی یک Bounded Context می شود که می توان آنها را یک Solution کد در ویژوال استادیو یا یک پروژه جداگانه در دیگر سکوها دید و اینها نقش سرویس ها را ایفا می کنند!

sub system == sub domains == bounded context (code solution) == micro service



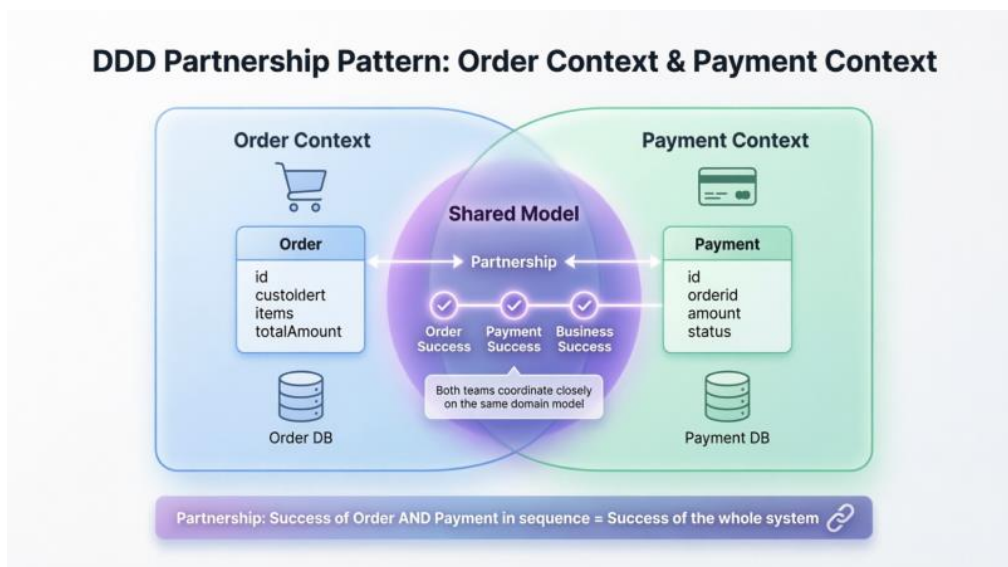
نگاشت زمینه ها (Context Mapping): وقتی چند Bounded Context در یک سازمان وجود دارند، باید رابطه بین اعضای آنها مشخص شود. Context Map ابزاری برای مستندسازی این روابط است. به بیانی دیگر، Context Map سندی است که رابطه بین Bounded Domain ها در آن تعریف می شود و نحوه ارتباط تیم ها منیج می شود.



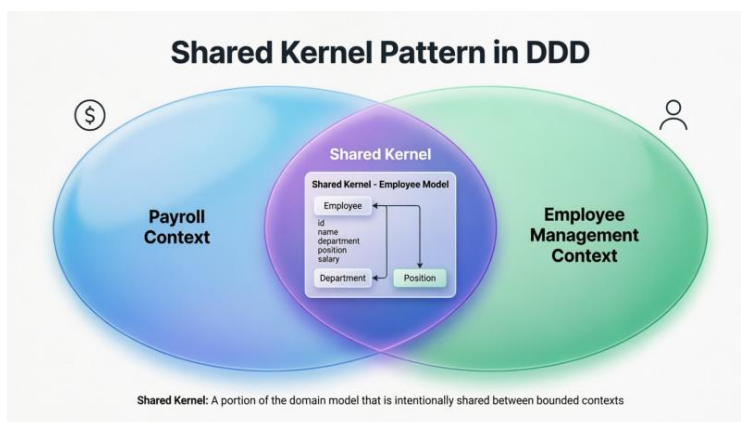
مدل های دامنه (Domain Models): مدل های دامنه، نمایانگر مفاهیم کسب و کار و ارتباطات بین آنها هستند. این مدل ها می توانند به صورت دیاگرام، مستند نوشته شده یا حتی کدی باشند.

مهم‌ترین الگوهای رابطه بین اعضای یک Bounded Context عبارتند از:

Partnership 🏠: دو تیم به صورت هم‌راستا و با هماهنگی مداوم کار می‌کنند. در این نوع رابطه، دو Bounded Context به‌شدت به هم وابسته‌اند و موفقیت یا شکست یکی روی دیگری اثر مستقیم دارد؛ بنابراین تیم‌های مسئول آن‌ها باید برای توسعه، تغییر API ها و Integration با هم هماهنگ باشند. این رابطه دوطرفه و متقارن است و Upstream/Downstream مشخصی ندارد. مثال: دو Bounded Context به نامهای Order Context و Payment Context که موفقیت هر دو، شرط مهم موفقیت کل روند ثبت Order است!

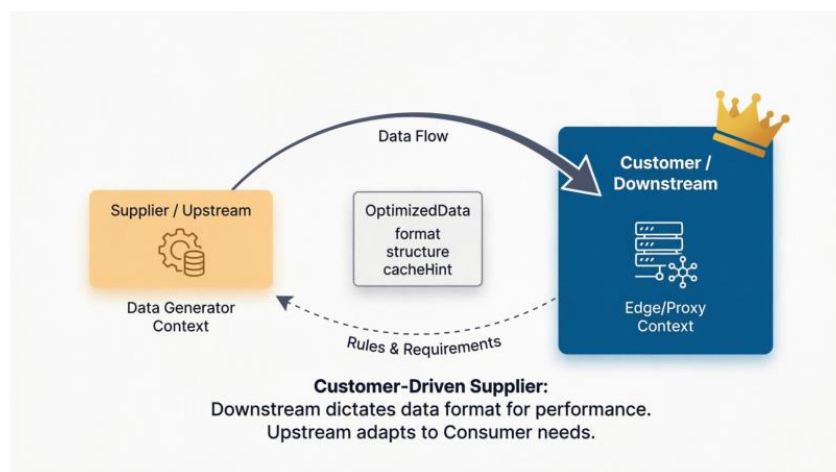


Shared Kernel 🏠: بخشی از مدل (کد یا داده) که بین دو یا چند Context به اشتراک گذاشته می‌شود و نیازمند هماهنگی نزدیک بین بخش‌های مشترک است! به بیان بهتر کرنل مشترک، به بخشی از مدل یا کد مشترک بین دو یا چند Bounded Context گفته می‌شود. هر تغییری در این بخش می‌تواند روی همه Context‌های استفاده‌کننده از آن تأثیر بگذارد؛ بنابراین تیم‌های مرتبط باید از تغییرات آن باخبر باشند و درباره آن با یکدیگر هماهنگ شوند. برای مثال، ممکن است چند بخش مختلف سیستم از یک تعریف مشترک برای Money یا همان پول استفاده کنند. در این حالت، تغییر در تعریف یا رفتار Money می‌تواند روی همه این بخش‌ها اثر بگذارد. به همین دلیل، تیم‌هایی که یک Shared Kernel دارند باید ارتباط و هماهنگی نزدیکی با یکدیگر داشته باشند. هرگاه قرار باشد تغییری در بخش مشترک ایجاد شود، تیم‌های درگیر باید قبل از اعمال تغییر، درباره آن هماهنگ شوند تا مطمئن شوند تغییر جدید باعث اختلال در Context‌های دیگر نمی‌شود.

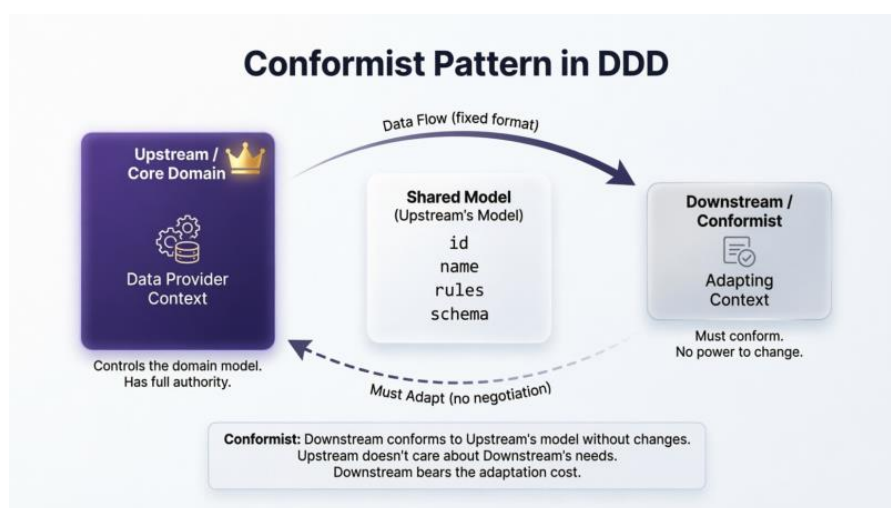


Customer-Supplier 🏠: یک Context به عنوان Supplier داده‌ای تولید می‌کند و دیگر Context به عنوان Customer از آن داده استفاده می‌کند و اصطلاحاً آن داده را مصرف می‌کند. در این رویه معمولاً Supplier تلاش می‌کند انتظارات و نیازهای Customer را بدون

مشکل آفرینی برآورده کند! لذا معمولاً Customer حق وتو و تعیین قوانین را دارد چون دیتای فراهم شده نباید موجب بوجود آمدن مشکلات برای آن شود. به این رابطه، رابطه up stream / down stream هم می گویند. در این مدل از دامین ها، هماهنگی بین تغییرات فراهم کننده و مصرف کننده مهم است. در این ساختار، مصرف کننده می تواند استراتژی و شیوه دیتا را عوض کند و فراهم کننده هم باید مطابق با مصرف کننده عوض شود. برای مثال در یک سیستم Proxy Server، وقتی لایه Edge به کلاینت های وب سایتها، خدمت رسانی می کند و شرط مهم، سرعت بالای خدمت رسانی به کلاینت ها است، می تواند در مورد نحوه و ساختار دیتایی که از بخش جنریت دیتا می گیرد اعمال نفوذ کند! چون جنریت دیتا کار ساده ای است که حجم بالای درخواست را سمت خود ندارد (فقط Api های سمت Edge که نهایتاً چند صد یا چند هزار سرور در نقاط مختلف جهان هستند، هر چند ثانیه یک بار، دیتای خود را به عنوان مصرف کننده، به روز می کنند)، ولی بخش پاسخگو به ریکوئست کاربران (Edge) به جهت ترافیک بالایی که روی خود دارد، باید تا حد امکان بهینه باشد و از دیتای بهینه به شکل کارآمد استفاده کند و تا حد امکان از محاسبات زیربنایی روی دیتا پرهیز کند و محاسبات زیربنایی را به فراهم کننده دیتا ارسال کند تا در زمان پاسخگویی، با کمترین محاسبات ممکن، سریع ترین کارایی را داشته باشد. لذا در اینجا، فراهم کننده باید تلاش کند خواسته های مصرف کننده دیتا را به جا آورد.



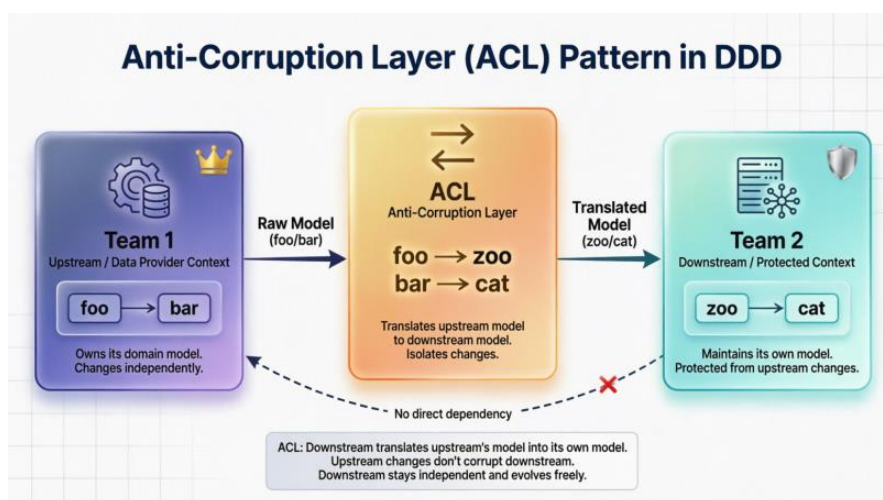
Conformist 🏠: برعکس حالت فراهم کننده و مصرف کننده، در این حالت، Context مصرف کننده مدل Context بالادستی را بدون تغییر می پذیرد؛ معمولاً چون قدرت تغییر آن را ندارد! پس در این وضعیت، Core Domain اطلاعات خودش را تغییر می دهد و به روز می کند و دغدغه به روز شدن بخش های دیگر را ندارد و آنها باید خودشان را اصلاح و به روز کنند.



Anti-Corruption Layer (ACL) 🏠 یا لایه زایل نشدنی یا ناگنبدیدی: یک لایه میانی که مدل یک سیستم خارجی و احتمالاً قدیمی را ترجمه می کند تا مدل داخلی اصطلاحاً آلوده نشود. این لایه باید شما را یاد الگوی Adapter و Bridge (مثال مترجم) بیندازد که از

پراکاربدرترین الگوها هستند! لایه ای بین دو SubDomain ایجاد می شود که تمام اصطلاحات لایه های دیگر را Map و ترجمه می کند تا اگر نام یکی از اجزا عوض شود، به راحتی، بخش های دیگر هم با آن هماهنگ شوند و نامشان دست نخورد. در واقع لایه های دیگر، نام آن را خودشان برای خودشان تعیین می کنند و صرفاً باید ولیوی کلید نام آن به نام جدید تبدیل شود.

پس به فرض شما دو میکروسرویس دارید که با هم مرتبط هستند ولی برای ارتباط، Adapter یا wrapper یا لایه مترجمی می گذارید که فارغ از زبان خودشان، به هر کدام، سرویس خاص خود را می دهد و زبان بینشان را ترجمه و بهینه سازی می کند و باعث می شود اگر ادبیات یکی عوض شد، ادبیات دیگری عوض نشود و ادبیات در همین لایه Adapter به روز می شود! مثلاً در یک بانک، Account Manager، در ارتباط با User Manager، از یک لایه Adapter استفاده کرده و اگر روزی، تغییری در جدول یا زبان Account Manager بوجود آید، تغییرات در Adapter اعمال می شود و در User Manager تغییری اعمال نمی شود چون Adapter مد نظر Data را قبلاً به زبان خاص User Manager در اختیار وی قرار داده است! [سیستم قدیمی Legacy] ← [ACL: تبدیل و پاکسازی داده] ← [سیستم جدید DDD]

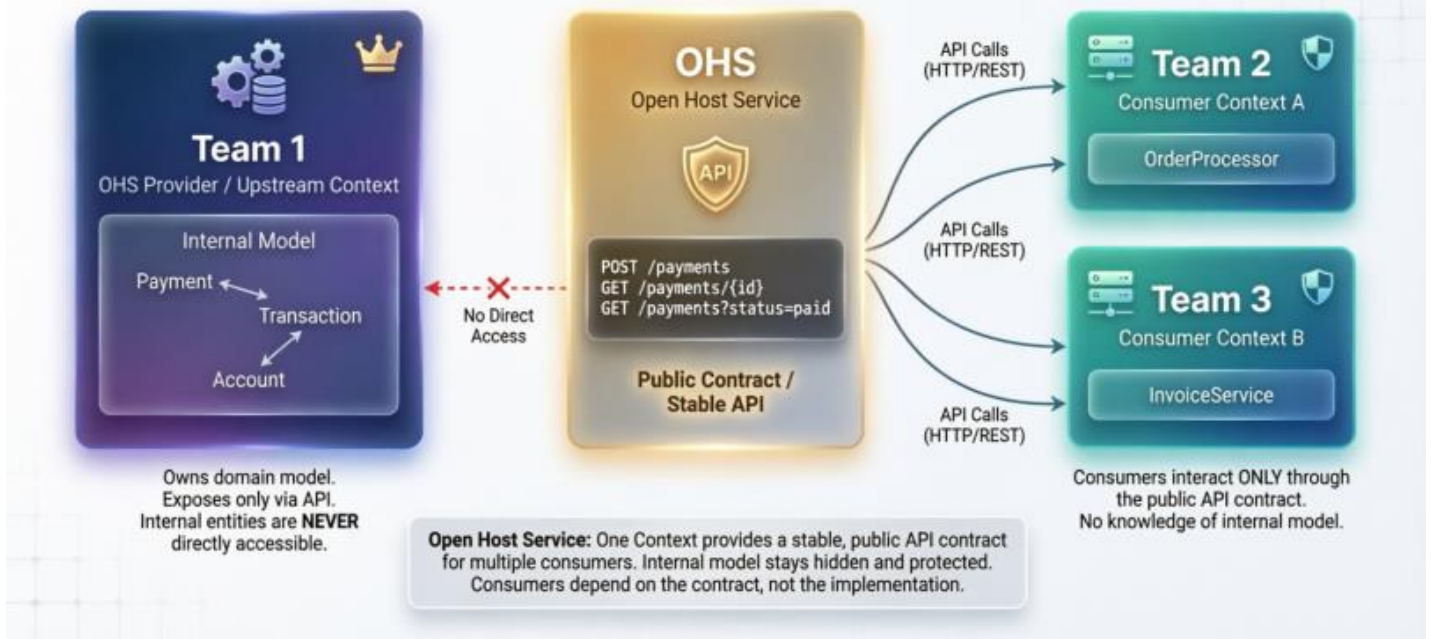


Open Host Service (OHS): یک Context، API عمومی و پایدار برای استفاده چند Context دیگر ارائه می دهد! OHS به نوعی Bounded Context است که یک مجموعه سرویس یا API عمومی و مشخص ارائه می دهد تا Bounded Context های دیگر بتوانند از قابلیت های آن استفاده کنند. به جای اینکه هر Context مستقیماً به مدل داخلی Context دیگر وابسته شود، OHS یک قرارداد (Contract) مشخص و پایدار در اختیار مصرف کنندگان قرار می دهد و همگان موظفند برای آن سرویس، از آن قرارداد عمومی استفاده کنند! مثلاً Payment Context می تواند API هایی مثل:

```
POST /payments
GET /payments/{id}
```

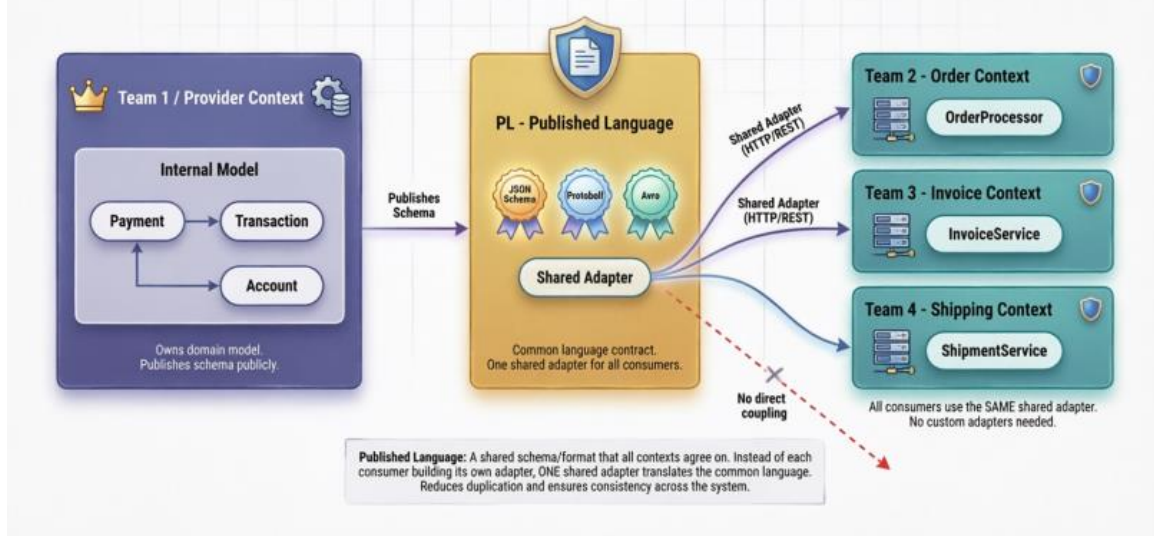
ارائه کند و Context های دیگر فقط باید با همین قرارداد عمومی ارتباط داشته باشند، بدون اینکه از Entity ها، Database یا منطق داخلی Payment Context اطلاعی داشته باشند. پس Open Host Service یعنی یک رابط عمومی برای تعامل با یک Bounded Context! نه اینکه مدل داخلی آن Context عمومی یا قابل دسترسی مستقیم باشد.

Open Host Service (OHS) Pattern in DDD

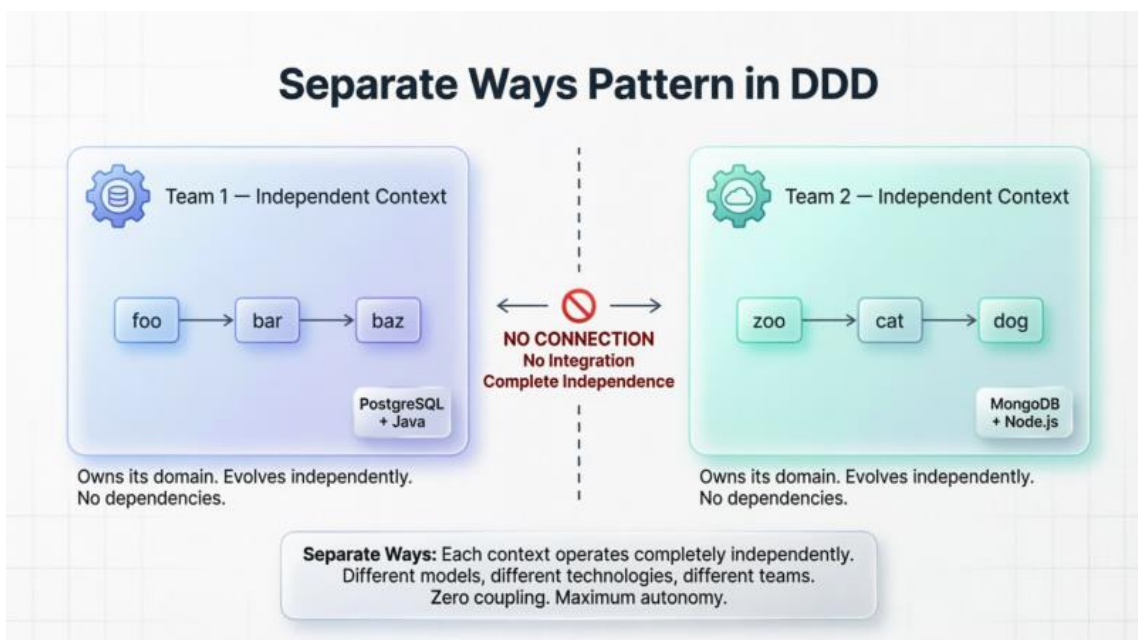


Published Language: یک زبان یا فرمت مشترک (مثل یک اسکیمای (Schema) استاندارد) برای تبادل داده بین Context ها. به بیانی دیگر، سرویس های مختلف به جای اینکه هر کدامشان، Adapter تکراری و اختصاصی خودشان را داشته باشند تا زبان دیگر سرویس ها را بدانند، Protocol سرویسی که به صورت مشترک مورد استفاده همه یا اغلب سرویس ها قرار می گیرد را توسط یک Adapter اشتراک داده شده به دست می آورند و خودشان را با آن هماهنگ می کنند! گاهی این Adapter می تواند مشترک بین همه باشد و به عبارت بهتر یک زبان مشترک بین تمام زیرسیستم ها باشد.

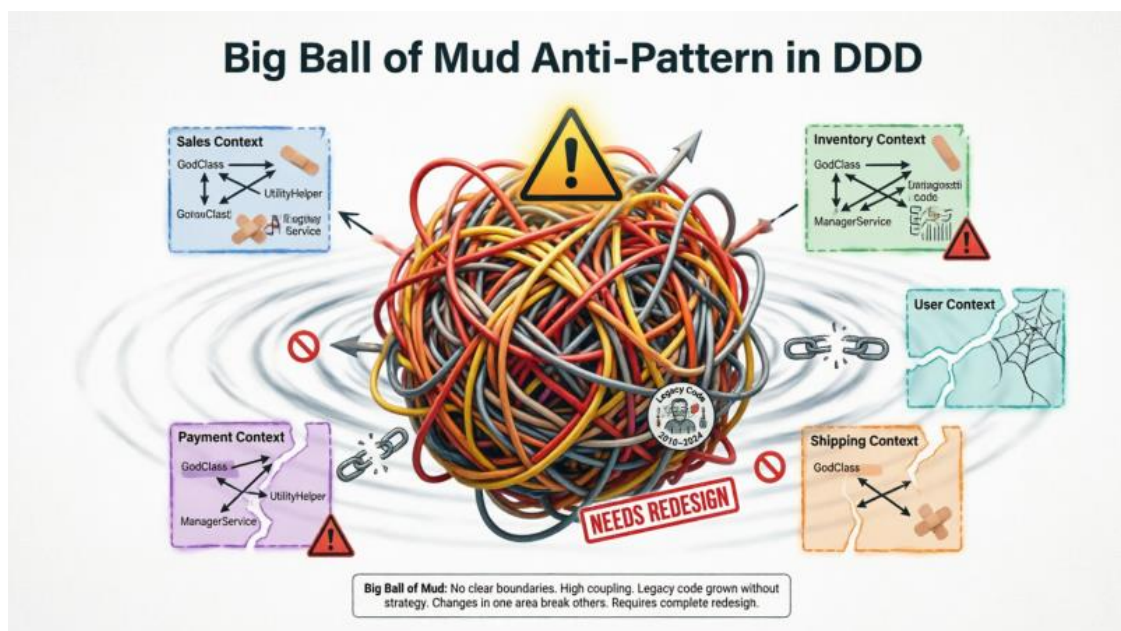
Published Language (PL) Pattern in DDD



Separate Ways یا مسیر های منفرد: دو سرویس یا Bounded Context یا Context هیچ ارتباط خاصی با هم ندارند و هر کدام مستقل عمل می کنند! پس این مورد به تیم های هر دو سرویس اجازه می دهد با روش و مکانیسم و تکنولوژی خاص خود آنها را پیش ببرند.



Big Ball of Mud: وضعیتی که مرزهای Context ها مشخص نیستند و سیستم به شکل درهم تنیده و بدون ساختار رشد کرده است و بهتر است از آن اجتناب شود! در چنین سیستم هایی گاه یک فرد یا نهاد وظایف بخش های مختلف را انجام می دهند (آچار فرانسه)! معمولاً کدهای قدیمی Legacy که چندین سال توسط افراد مختلف بدون هیچ طرح و استراتژی خاصی توسعه یافته باشد، به این شکل تبدیل می شود! کلاسهای چند هزار خطی، وابستگی های شدید بین اعضا، وظایفی از برخی زیرسیستم ها که در بخش های دیگر انجام می شود، کدهای وصله پینه، با دست بردن برای تعمیر یک بخش، عملکرد بخش های دیگر نیز دچار تغییراتی می شوند! این وضعیت، کار توسعه نرم افزار را به سمت نیازمند شدن به Redesign سوق می دهد! این وضعی است که در برخی Bounded Context ها هم شاهد هستیم! به بیان بهتر وضعیتی است که در برخی ارگانهای یک بیزنس، شاهد آن هستیم! زیربخش هایی بدون مرزبندی های دقیق و افراد چند وظیفه و ...





طراحی تاکتیکی (Tactical Design): طراحی تاکتیکی به این سؤال پاسخ می دهد: «داخل یک Bounded Context، کد را چگونه بنویسیم؟» این بخش شامل الگوهای بلوک ساز (Building Blocks) زیر است:



Entity (موجودیت): Entity شیء ای است که با یک شناسه و هویت یکتا (Identity) تعریف می شود، نه با مقادیر ویژگی هایش. دو Entity حتی اگر تمام مشخصاتشان یکسان باشد اما ID متفاوت داشته باشند، متفاوت محسوب می شوند. Entity ها معمولاً در طول زمان تغییر می کنند اما هویتشان (ID) ثابت می ماند! پس Entity ها در طول زمان حتی با تغییر ویژگی هایشان، همان شیء باقی می ماند. مانند یک انسان که با تغییر در وزن و قد و سن و تخصصش، همان انسان باقی مانده است! یا یک Order با تغییر مشخصاتش، همان Order باقی مانده است! یا حتی اگر دو رکورد در دو جای مختلف با یک Id داشته باشیم و بخواهیم آنها را با هم Integrate کنیم، مجبوریم یک Order حساب کنیم و بر اساس تاریخ آخرین تغییرات (رکوردی که فیلد Update_Date اش جدیدتر است)، مشخصاتش را به روز کنیم!

```
#include <string>
#include <utility>
class Customer {
private:
    CustomerId id_;           // هویت یکتا
    std::string name_;
    Email email_;

public:
    Customer(
        CustomerId id,
        std::string name,
        Email email)
        : id_(std::move(id)),
          name_(std::move(name)),
          email_(std::move(email)) {}

    const CustomerId& getId() const {
        return id_;
    }
    const std::string& getName() const {
        return name_;
    }

    const Email& getEmail() const {
        return email_;
    }

    void changeEmail(const Email& newEmail) {
        email_ = newEmail;
    }
};
```



Value Object (شیء دارای ارزش خاص): برخلاف Entity، یک Value Object با مقادیر ویژگی هایش تعریف می شود، نه با هویتش. پس دو Value Object با مقادیر یکسان، برابر تلقی می شوند. Value Object ها باید تغییرناپذیر (Immutable) باشند، یعنی به جای تغییر مقدار، یک نمونه جدید ساخته می شود. در کد زیر، به متد add توجه کنید که یک شیء جدید از نوع Money را می سازد و مقدار Add شده را به شکل یک شیء جدید return می کند! Value Object ها بدون هویت مستقل؛ فقط بر اساس مقادیرش تعریف می شود!

```

#include <stdexcept>
#include <string>
#include <utility>

class Money {
private:
    double amount_; // مثل 25.50
    std::string currency_; // مثل ۵۰۰ یا دلار یا یورو

public:
    Money(double amount, std::string currency)
        : amount_(amount),
          currency_(std::move(currency)) {
        if (amount_ < 0) {
            throw std::invalid_argument(
                "مبلغ نمیتواند منفی باشد"
            );
        }
    }

    double getAmount() const {
        return amount_;
    }

    const std::string& getCurrency() const {
        return currency_;
    }

    Money add(const Money& other) const {
        if (other.currency_ != currency_) {
            throw std::invalid_argument(
                "واحد پول ناسازگار است"
            );
        }

        return Money(
            amount_ + other.amount_,
            currency_
        );
    }

    bool operator==(const Money& other) const {
        return amount_ == other.amount_ &&
            currency_ == other.currency_;
    }

    bool operator!=(const Money& other) const {
        return !(*this == other);
    }
};

```

🎨 **مثال‌های رایج Value Object (شیء مقداری):** آدرس پستی، بازه زمانی، مبلغ پول، مختصات جغرافیایی، ایمیل. برای توضیح بیشتر، دو آدرس منزل با مقادیر برابر، به یک منزل اشاره می‌کند و یکی هستند! ابعاد دو گوشی که مدلشان برابر است، یکیست! مختصات طول و عرض جغرافیایی به یک جا اشاره می‌کند حتی اگر هزاران مختصات طول و عرض جغرافیایی دارای طول و عرض برابر باشند!

Invariant  قانون یا شرطی که در مورد یک Value Object باید همیشه برقرار باشد و نقض نشود! برای مثال در مورد Money یا پول، مبلغ همیشه باید بزرگتر یا مساوی 0 باشد و Currency پول اضافه شونده به پول قبلی باید همیشه از نوع Currency قبلی باشد! (ریال به روی ریال اضافه یا از آن کم می شود! نه دلار اضافه شونده به ریال)

```
amount >= 0
currency != null
```

Invariantهای یک Product 

```
Color (RGB)
Code (Country Code + Type + Index) نباید بیشتر از 7 رقم شود
Name نباید خالی باشد
Price > 0
Type (10,20,30)
Country Code باید 3 رقمی باشد!
```

Invariantهای یک Customer 

```
National Code باید 10 رقمی باشد و مثلاً یکتا باشد
Age > 0
Name نباید خالی باشد
```

Anemic vs. Rich Domain Model 

Anemic Domain Model  (ضدالگو ) Entityها فقط Getter/Setter دارند و تمام منطق در Service Layer است. به بیان بهتر Anemic Domain Model مدلی است که اشیای دامنه عمدتاً فقط State/Data را نگه می دارند و رفتارها و Business Ruleهای مهم دامنه عمدتاً خارج از آنها، مثلاً در Serviceها، قرار گرفته اند.

```
class Order {
private:
    OrderStatus status_;

public:
    OrderStatus getStatus() const {
        return status_;
    }

    void setStatus(OrderStatus status) {
        status_ = status;
    }
};

class OrderService{
public:
    void cancelOrder(Order& order) {
        if (order.getStatus() == OrderStatus::Shipped) {
            throw std::logic_error(
                "Cannot cancel shipped order"
            );
        }

        order.setStatus(OrderStatus::Cancelled);
    }
}
```

Rich Domain Model  (الگوی درست ) : منطق کسب و کار داخل خود Entity/Aggregate قرار دارد. DDD همواره طرفدار Rich Domain Model است! یعنی منطق باید داخل خود Entity/Aggregate باشد. به بیان بهتر هر Entity روند خود کنترلی را در ورودی ها

دارد و به جای اینکه کنترلر جداگانه ای، ورودی های آنرا کنترل و Invariant ها را مهندسی کند، متد های Setter خود مدل Entity، آنها را کنترل می کند و داده نامعتبر را شناسایی و مهندسی می کند!

```
#include <stdexcept>

class Order {
private:
    OrderStatus status;

public:
    void cancel()
    {
        if (status == OrderStatus::Shipped) {
            throw std::logic_error(
                "Cannot cancel shipped order"
            );
        }

        status = OrderStatus::Cancelled;
    }
};
```

Aggregate و **Aggregate Root**   خوشه ای از Entity ها و Value Object ها است که به عنوان یک واحد یکپارچه در نظر گرفته می شود و قوانین سازگاری (Consistency Rules / Invariants) بین اعضای آن باید همیشه برقرار باشد.

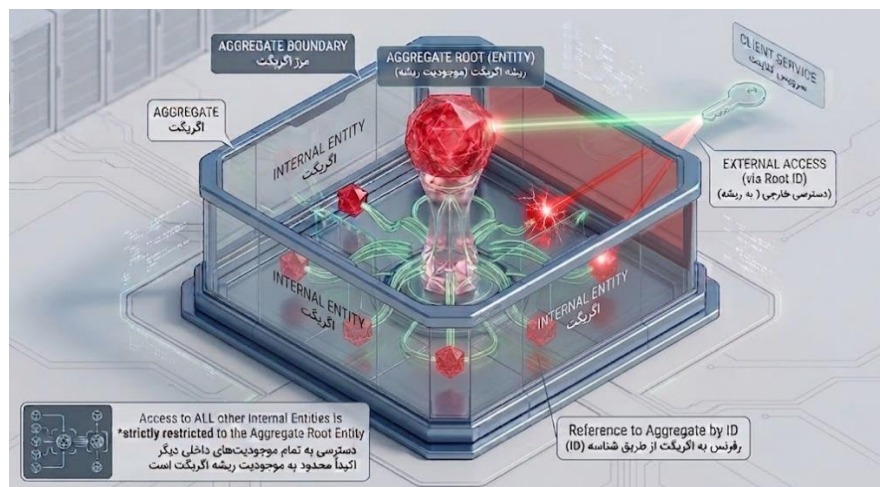
می توان یک Aggregate را به شکل یک Struct با داشتن مجموعه ای از Value Object ها (شبه متغیرهای primitive در struct) و Entity ها (شبه اشیای پیچیده تر داخل struct که هویتشان با id مشخص می شود) در نظر گرفت! باید سعی کنید Aggregate ها را تا حد امکان کوچک و لاغر نگه دارید و به نوعی Single Responsibility Principle از قوانین Solid را روی آنها رعایت کنید!

هر Aggregate یک Aggregate Root دارد؛ یعنی یک شیء که تنها نقطه ورود دنیای بیرون به Aggregate است. دنیای بیرون برای تعامل با Aggregate باید از طریق Root اقدام کند و نباید مستقیماً به اشیای داخلی Aggregate دسترسی داشته باشد یا آنها را تغییر دهد. به بیان دیگر، اشیای داخل Aggregate به جز Root بخشی از مرز داخلی آن هستند و مسئولیت حفظ سازگاری و قوانین دامنه بر عهده Root است و فقط Aggregate Root از بیرون قابل ارجاع است و فقط از طریق Root و با مدیریت آن می توان در سیستم تغییر ایجاد کرد!

برای درک این مفهوم در C++ می توان یک مدل ذهنی ساده در نظر گرفت: تصور کنید یک struct یا class بیرونی، تعدادی شیء داخلی از انواع پیچیده و Sub System (اشیای ساخته شده از کلاس های کوچکتر که بخشی از کارهای Aggregate Root را Handle می کنند) را در خود به صورت private نگه می دارد. کدهای بیرون فقط با شیء بیرونی تعامل می کنند و برای انجام یک عملیات، درخواست خود را به آن می دهند. شیء بیرونی نیز در صورت نیاز، عملیات را به اشیای داخلی واگذار می کند و آنها هرکدام بخشی از کار را انجام می دهند تا در نهایت عملیات موردنظر توسط Aggregate Root به صورت نهایی و کامل در می آید و نتیجه به شیء درخواست کننده (به طریقی) داده می شود.

از این منظر، Aggregate Root را می توان از نظر مدل ذهنی تا حدودی شبیه یک Facade در نظر گرفت! لازم نیست اشیای بیرون Aggregate Root بدانند اشیای داخلی چگونه با یکدیگر همکاری می کنند؛ فقط با Root صحبت می کنند و Root هماهنگی های لازم را انجام می دهد.

البته این تشبیه صرفاً برای ساختن یک مدل ذهنی در C++ است! Aggregate در DDD الزاماً یک struct تودرتو یا حتی یک ساختار داده مشخص نیست؛ مهم تر از شکل ظاهری آن، مرز Aggregate، نقطه ورود مشخص و حفظ قوانین و سازگاری دامنه درون این مرز است.



```
#include <vector>
#include <string>
#include <utility>
#include <stdexcept>
// قبلاً تعريف شده اند Product و ProductId فرض بر اين است كه
// =====
// Entity داخل Aggregate
// =====
class OrderLine{
private:
    ProductId productId_;
    int quantity_;
    // فقط Order مي‌تواند OrderLine بسازد
    friend class Order; // براي اينكه دسترسی خصوصی به کلاس اوردر دهيم

    OrderLine(
        const ProductId& productId,
        int quantity)
        : productId_(productId),
          quantity_(quantity) {}

public:
    const ProductId& getProductId() const {
        return productId_;
    }
    int getQuantity() const {
        return quantity_;
    }
};
// =====
// Aggregate Root
// =====
class Order {
private:
    OrderId id_;
    // فقط Order به OrderLine ها دسترسی مستقيم دارد
    std::vector<OrderLine> orderLines_;
public:
    explicit Order(const OrderId& id)
        : id_(id) {}
    void addItem(
        const Product& product,
        int quantity) {}
};
```

```

    منطق کسبوکار و validation اینجا اجرا می‌شود
    if (quantity <= 0){
        throw std::invalid_argument(
            "Quantity must be greater than zero"
        );
    }
    orderLines_.emplace_back(
        product.getId(),
        quantity
    );
}
const OrderId& getId() const {
    return id_;
}
const std::vector<OrderLine>& getOrderLines() const {
    return orderLines_;
}
};

```

🤖 مقایسه Aggregate و Entity: گاهی تفکیک تفاوت بین Aggregate و Entity سخت می‌شود!

برای توضیح بهتر، تفاوت بینشان را بیان می‌کنیم!

- Entity هایی مثل OrderItem به خودی خود تصمیم‌گیر محاسبات داخلی خود نیست و نمی‌توانند مستقلاً تغییر کنند.
- و همیشه بخشی از یک Aggregate بزرگ‌تر (Order) است.
- و قانون‌هایش در دل Aggregate اصلی یعنی Order بررسی می‌شوند (مثلاً افزایش تعداد یا حذف فقط از طریق Order ممکن است!)

💡 نکته کلیدی:

هر Aggregate می‌تواند شامل یک یا چند Entity باشد، ولی هر Entity لزوماً و همیشه یک Aggregate نیست.

🔗 برای تبدیل یک Entity به یک Aggregate :

1. آیا نیاز به مرز مدیریت consistency مستقل دارد؟
2. آیا قوانین تجاری مربوط به خودش را خودش کنترل می‌کند؟
3. آیا می‌تواند بدون تکیه به موجودی دیگر، در سیستم رفتار داشته باشد؟

اگر پاسخ همه «بله» بود ← Aggregate Root

اگر پاسخ «نه» بود ← یک Entity ساده است!

💡 قوانین طلایی Aggregate:

🔗 رابطه Aggregate و Transaction: یکی از مهم‌ترین نکات درباره Aggregate این است که Aggregate را مرز

سازگاری (Consistency Boundary) در نظر می‌گیریم. یعنی چه؟

هر **Aggregate** به عنوان یک مرز سازگاری عمل می کند. قوانین تجاری درون یک **Aggregate** باید همیشه و به صورت اتمیک (Atomic) حفظ شوند؛ به این معنی که در قالب یک واحد کاری (Transaction-Unit of Work)، یا تمام تغییرات درون **Aggregate** با موفقیت اعمال می شوند یا هیچ تغییری ثبت نمی شود.

اگرچه از نظر فنی می توان یک تراکنش پایگاه داده را روی چند **Aggregate** اجرا کرد، اما در **DDD** معمولاً هر **Transaction** یک **Aggregate** را به عنوان واحد **Consistency** تغییر می دهد و تا حد امکان از تغییر همزمان چند **Aggregate** در یک **Transaction** پرهیز می شود و تعاملات بین چند **Aggregate** از طریق سازگاری تدریجی یا **Eventual Consistency** مدیریت شود (یعنی با استفاده از **Domain Event + Event Handler/Subscriber**).

- **تمرکز تراکنش:** تراکنش های پایگاه داده باید محدود به یک **Aggregate** باشند؛ از دستکاری همزمان چند **Aggregate** در یک تراکنش تا حد امکان پرهیز کنید.

- **کوچک نگه داشتن ابعاد:** **Aggregate** ها باید تا حد امکان کوچک طراحی شوند تا نرخ تعارضات همزمانی یا **Concurrency Conflicts** کاهش یابد و عملکرد سیستم بهینه تر شود.

- **ارتباط از طریق شناسه (ID):** ارجاع بین **Aggregate** های مختلف هرگز نباید به صورت مستقیم (Object Reference) باشد، بلکه صرفاً باید از طریق شناسه (ID) انجام پذیرد.

پس ما عملاً می توانیم عملیات بین چند **Aggregate** را در یک **Database Transaction** انجام دهیم و همه تغییرات با هم **Commit** شوند یا در صورت بروز خطا، همه آن ها **Rollback** شوند. ولی مشکل از جایی شروع می شود که همیشه و به طور مداوم مجبور باشیم چند **Aggregate** جدا از هم را در یک **Transaction** و به صورت اتمیک تغییر دهیم. در چنین شرایطی باید از خودمان بپرسیم:

آیا این **Aggregate** ها واقعاً باید از یکدیگر جدا باشند؟ چون اگر برای حفظ یک قانون کسب و کار مجبور باشیم همیشه چند **Aggregate** را همزمان تغییر دهیم، ممکن است مرز **Aggregate** ها را اشتباه انتخاب کرده باشیم. بنابراین بهتر است یک مدل ذهنی ساده داشته باشید:

هر **Aggregate** مسئول حفظ سازگاری و قوانین داخلی خودش است. یک **Use Case** می تواند با چند **Aggregate** کار کند، اما نباید برای حفظ قوانین داخلی یک **Aggregate**، دائماً به تغییر همزمان چند **Aggregate** وابسته باشیم.

در نتیجه، یک قانون مهم در معماری **Aggregate** ها آنست که بگوییم **Aggregate** باید مرز **Consistency** باشد.

💡 **قانون مهم:** یک **Aggregate** یک سیستم کامل و مستقل با داشتن یک **Aggregate Root** است که می تواند شامل زیر سیستم های وابسته به خودش به عنوان **Entity** ها باشد و صفر تا صد یک بیزنس را به صورت مستقل مدیریت می کند و یا همه عملیاتش انجام می شوند و یا در صورت وقوع مشکلی، تمام عملیات **Roll Back** می شوند! (**Unit Of Work – Transaction**) (بالاتر مطرح شد، به جهت اهمیت موضوع، بار دیگر تکرار شد!)

تبادل داده بین Bounded Context ها: هر **Bounded Context** آداپتورهای خارجی (**External Adapters**) اختصاصی خود را تنظیم می کند تا داده های مشترک و مورد نیاز سایر بخش ها را به اشتراک بگذارد. سرویس های دیگر نیز با اتکا به همین آداپتورها و داده های ارائه شده، منطق و پیاده سازی خاص خود را انجام می دهند.

ثبت سفارش

|

— Order تغییر

— Inventory تغییر

— Payment ثبت



Domain Event (رویداد دامنه): Domain Event واقعه ای است که در دامنه اتفاق افتاده (زمان گذشته) و برای بخش‌های دیگر سیستم اهمیت دارد؛ مثل OrderConfirmed یا PaymentReceived!

Domain Event ها امکان کاهش وابستگی مستقیم بین Aggregate ها و Bounded Context ها را فراهم می‌کنند و پایه معماری‌های Event-Driven هستند. به بیان خیلی ساده تر، به جای اینکه اجرای یک یا چند متد از یک یا چند Aggregate را منوط به اجرای یک تابع دیگر کنید و برای این کار، به سبک گذشته، اشیای کلاسهای مدنظرتان را به آن کلاس یا Aggregate پاس دهید، یک Event Bus یا Event Sourcing، رویدادهای Dispatch شده از یک کلاس Aggregate را به یک یا چند Aggregate که منتظر آن رویداد بودند، متصل می‌کند! این روش وابستگی موجودیتی و Coupling بین اشیاء را از بین می‌برد! در واقع الگوی حلال این روش، الگوی Observer و همچنین الگوی Publish/Subscriber است!

مثال کامل:

```
#include <chrono>
#include <memory>
#include <vector>
#include <string>
#include <stdexcept>
#include <utility>

namespace domain::order {
// =====
// Types & Identifiers
// =====
using OrderId = std::string;
using CustomerId = std::string;
using ProductId = std::string;
```

```

class DomainException : public std::runtime_error {
public:
    explicit DomainException(const std::string& message)
        : std::runtime_error(message) {}
};

enum class OrderStatus {
    Draft,           // پیش‌نویس
    Placed,          // ثبت‌شده
    Confirmed        // تأییدشده
};

// =====
// Value Object: Money (تغییرناپذیر و دارای اعتبارسنجی)
// =====

class Money {
private:
    double amount_;
    std::string currency_;

public:
    Money(double amount, std::string currency)
        : amount_(amount),
          currency_(std::move(currency)) {
        if (amount_ < 0) {
            throw std::invalid_argument("مبلغ نمی‌تواند منفی باشد.");
        }
    }

    double getAmount() const {
        return amount_;
    }

    const std::string& getCurrency() const {
        return currency_;
    }

    Money add(const Money& other) const {
        if (other.currency_ != currency_) {
            throw std::invalid_argument("واحد پول ناسازگار است.");
        }

        return Money(amount_ + other.amount_, currency_);
    }

    bool operator==(const Money& other) const {
        return amount_ == other.amount_ && currency_ == other.currency_;
    }

    bool operator!=(const Money& other) const {
        return !(*this == other);
    }
};

// =====
// Domain Events
// =====

class IDomainEvent {
public:
    virtual ~IDomainEvent() = default;
    virtual std::chrono::system_clock::time_point getOccurredOn() const = 0;
};

```

```

class OrderPlacedEvent : public IDomainEvent {
private:
    OrderId orderId_;
    CustomerId customerId_;
    std::chrono::system_clock::time_point occurredOn_;

public:
    OrderPlacedEvent(const OrderId& orderId, const CustomerId& customerId)
        : orderId_(orderId),
          customerId_(customerId),
          occurredOn_(std::chrono::system_clock::now()) {}

    const OrderId& getOrderId() const { return orderId_; }
    const CustomerId& getCustomerId() const { return customerId_; }
    std::chrono::system_clock::time_point getOccurredOn() const override {
        return occurredOn_; }
};

class OrderConfirmedEvent : public IDomainEvent {
private:
    OrderId orderId_;
    std::chrono::system_clock::time_point occurredOn_;

public:
    explicit OrderConfirmedEvent(const OrderId& orderId)
        : orderId_(orderId),
          occurredOn_(std::chrono::system_clock::now()) {}

    const OrderId& getOrderId() const { return orderId_; }
    std::chrono::system_clock::time_point getOccurredOn() const override { return occurredOn_; }
};
// =====
// Internal Entity (موجودیت داخلی)
// =====
class Order; // Forward Declaration
// چون هنوز اوردور تعریف نشده، اینجا کامپایلر درک می کند که بعدا تعریف خواهد شد

class OrderLine {
private:
    ProductId productId_;
    int quantity_;
    Money unitPrice_;
    friend class Order; // اجازه دسترسی کلاس اوردور به بخش خصوصی این کلاس

    OrderLine(const ProductId& productId, int quantity, const Money& unitPrice)
        : productId_(productId),
          quantity_(quantity),
          unitPrice_(unitPrice) {}

public:
    const ProductId& getProductId() const { return productId_; }
    int getQuantity() const { return quantity_; }
    const Money& getUnitPrice() const { return unitPrice_; }
};
// =====
// Aggregate Root
// =====
class Order {
private:
    OrderId id_;
    CustomerId customerId_;
    std::vector<OrderLine> lines_;

```

```

OrderStatus status_ = OrderStatus::Draft;
std::vector<std::shared_ptr<IDomainEvent>> domainEvents_;

void ensureDraft() const {
    if (status_ != OrderStatus::Draft) {
        throw DomainException("این عملیات فقط در وضعیت پیش‌نویس مجاز است.");
    }
}

public:
Order(const OrderId& id, const CustomerId& customerId)
    : id_(id), customerId_(customerId) {}
// Getters
const OrderId& getId() const { return id_; }

const CustomerId& getCustomerId() const { return customerId_; }

OrderStatus getStatus() const { return status_; }

const std::vector<OrderLine>& getLines() const { return lines_; }

const std::vector<std::shared_ptr<IDomainEvent>>& getDomainEvents() const {
    return domainEvents_; }
// Domain Methods
void addLine(const ProductId& productId, int quantity, const Money& unitPrice) {
    ensureDraft();

    if (quantity <= 0) {
        throw DomainException("تعداد محصول باید یک عدد مثبت باشد.");
    }
    lines_.emplace_back(productId, quantity, unitPrice);
}

void place() {
    ensureDraft();

    if (lines_.empty()) {
        throw DomainException("امکان ثبت سفارش خالی وجود ندارد.");
    }

    status_ = OrderStatus::Placed;

    domainEvents_.push_back(
        std::make_shared<OrderPlacedEvent>(id_, customerId_)
    );
}

void confirm() {
    if (lines_.empty()) {
        throw DomainException("سفارش خالی قابل تأیید نیست.");
    }
    status_ = OrderStatus::Confirmed;

    domainEvents_.push_back(
        std::make_shared<OrderConfirmedEvent>(id_)
    );
}

void clearDomainEvents() {
    domainEvents_.clear();
}

};

} // namespace domain::order

```

Repository انتزاع و الگویی است که دسترسی به Aggregate ها را شبیه یک مجموعه در حافظه فراهم می کند (Collection Like Interface) و جزئیات ذخیره سازی (پایگاه داده) را از دامنه پنهان می کند. ما در بخش الگوی Repository با این نوع ابزار آشنا شده ایم! البته لزومی ندارد که حتما به صورت مجموعه ای در حافظه نگه دارد! بلکه می توان مستقیما به دیتابیس خاصی متصل کرد! زیبایی کار با Repository در آنست که اگر قصد داشتید نوع دیتابیس پروژه را عوض کنید، کافیسست از تمام کلاسهای Repository قبلی در مسیری جدید مثل postgresql یا sqlite یا mysql کپی کنید و سپس query های قبلی دیتابیس را روی دیتابیس جدید تغییر دهید! با این کار به سادگی توان این را خواهید داشت که با تغییر در کانفیگ، Repository های خودتان را از مسیر دیگری بخوانید تا روی دیتابیس بعدی Shift کنید!

```
#include <memory>
#include <unordered_map>
#include <string>
#include <stdexcept>
//فرض: تعریف پایه ای OrderId و Order جهت کامپایل
using OrderId = std::string;

class Order {
private:
    OrderId id_;
public:
    explicit Order(OrderId id) : id_(std::move(id)) {}

    const OrderId& getId() const { return id_; }
};

class IOrderRepository {
public:
    virtual ~IOrderRepository() = default;

    virtual std::shared_ptr<Order> getById(const OrderId& id) = 0;

    virtual void add(const std::shared_ptr<Order>& order) = 0;

    virtual void remove(const std::shared_ptr<Order>& order) = 0;

    virtual void saveChanges() = 0;
};

// =====
// Implementation (حافظه در حافظه)
// =====
class InMemoryOrderRepository : public IOrderRepository {
private:
    std::unordered_map<OrderId, std::shared_ptr<Order>> storage_;

public:
    std::shared_ptr<Order> getById(const OrderId& id) override {
        auto it = storage_.find(id);
        if (it != storage_.end()) {
            return it->second; // بازگرداندن اشاره گر به سفارش
        }
        return nullptr; // اگر پیدا نشد
    }

    void add(const std::shared_ptr<Order>& order) override {
        if (!order) {
            throw std::invalid_argument("null وجود ندارد");
        }
    }
};
```

```

        storage_[order->getId()] = order;
    }

    void remove(const std::shared_ptr<Order>& order) override {
        if (order) {
            storage_.erase(order->getId());
        }
    }

    void saveChanges() override {
        // پیاده‌سازی In-Memory نیازی به عملیات خاصی نیست
        // در آداپتورهای واقعی مثل Entity Framework یا SQLite/Postgres
        // این متد نقش الگوی Unit of Work برای اجرای Commit را ایفا می‌کند
    }
};

```

🔦 قانون مهم: یک Repository فقط برای Aggregate Root تعریف می‌شود! نه برای هر Entity. برای مثال اگر هر Product به عنوان AggregateRoot دارای Aggregate داخلی مثل Image است و Image بدون Product معنی خاصی ندارد و با آن معنی می‌دهد! در این شرایط یک Repository برای Product تعریف می‌شود که هم خود Orderها را مدیریت می‌کند و هم Images را مدیریت می‌کند و لزومی ندارد یک Repository مثل OrdersRepository فقط برای مدیریت یک Table مثل Orders طراحی شود!

🏠 Factory (کارخانه)

Factory مسئول ساخت اشیای پیچیده (به‌ویژه Aggregateها) است، به‌خصوص وقتی ساخت شی نیازمند قوانین اعتبارسنجی یا هماهنگی چند بخش باشد. این کار منطق ساخت را از خود Entity جدا نگه می‌دارد. شاید یکی از اصلی‌ترین مزایای Factory این باشد که وقتی با پیاده‌سازی یک کلاس توسط Dependency Injection، در Aggregate Root به صورت مستقیم به خود Aggregateهای داخلی دسترسی ندارید تا از آنها نمونه‌سازی کنید (Include نکرده اید و فقط به Interface هایشان دسترسی دارید)، می‌توانید کار نمونه‌سازی را به عهده شی پاس داده شده از نوع Factory محول کنید! به بیان بهتر فرض کنید AggregateRoot تمام Aggregateهای داخلی را به صورت Interface می‌شناسد! حال اگر قرار باشد آنها را خودش از روی دیتابیس و ... بسازد، نمی‌تواند Interfaceها را new کند، ولی می‌تواند به شی پاس داده شده از نوع Factory بگوید یک نمونه از Aggregate داخلی با این مشخصات برایم بساز تا بسازد! به این شکل اصل Dependency Injection هم نقض نمی‌شود! به صورت استاندارد، یک کلاس Factory می‌تواند تمام کلاس‌هایی را که می‌تواند از آنها نمونه نو بسازد را Include کند!

```

#include <utility>
class OrderFactory {
public:
    static Order createFromCart(
        const ShoppingCart& cart,
        const Customer& customer) {
        Order order(
            OrderId::generate(),
            customer.getId()
        );

        for (const auto& item : cart.getItems()) {
            order.addItem(
                item.getProduct(),
                item.getQuantity()
            );
        }
        return order;
    }
};

```

Domain Service (سرویس دامنه):

وقتی منطقی وجود دارد که به طور طبیعی به یک Entity یا Value Object خاص تعلق ندارد (معمولاً چون چند Aggregate را درگیر می‌کند و بین چند Aggregate عمل می‌کند)، از Domain Service استفاده می‌شود. Domain Service بدون حالت (Stateless) است و فقط منطق خالص کسب‌وکار را دربر می‌گیرد. مثالی از Domain Service درگیر با چند Aggregate خود Order است! یک Order می‌تواند درگیر با این Aggregate Rootها باشد:

شهر و استان (استان به عنوان Aggregate Root و شهر به عنوان Aggregate داخلی استان)

کنتگوری‌ها (Aggregate Root کنتگوری‌ها)

یوزرها (Aggregate Root یوزرها)

```
class TransferService { // Domain Service
public:
    void transfer(
        Account& from,
        Account& to,
        const Money& amount) {
        from.withdraw(amount);
        to.deposit(amount);
    }
};
```

Application Service و Repository Interface  لایه‌ای که بین دنیای بیرون (کنترلر، API) و لایه دامنه قرار می‌گیرد. Application Service هیچ منطق کسب‌وکاری ندارد؛ فقط جریان کار (Workflow) را هماهنگ می‌کند! دریافت درخواست، فراخوانی Repository، فراخوانی متدهای Aggregate و ذخیره تغییرات.

```
#include <memory>
#include <stdexcept>
#include <string>
// =====
// Repository Interface
// تعریف شده در Domain Layer
// =====
class IOrderRepository {
public:
    virtual ~IOrderRepository() = default;

    virtual std::shared_ptr<Order> findById(const std::string& id) = 0;

    virtual void save(const std::shared_ptr<Order>& order) = 0;
};
// =====
// Application Service / Command Handler
// Use Case کننده هماهنگ
// =====
class ConfirmOrderCommandHandler {
private:
    std::shared_ptr<IOrderRepository> repository;
public:
    explicit ConfirmOrderCommandHandler(std::shared_ptr<IOrderRepository> repository)
        : repository(std::move(repository)) {}
};
```

```

void handle(const ConfirmOrderCommand& command) {
    auto order = repository->getById(command.getOrderid());
    if (!order)
    {
        throw std::runtime_error("سفارش یافت نشد");
    }
    //منطق کسب‌وکار داخل Aggregate است
    order->confirm();
    repository->saveChanges();
}
};

```

Application Layer شامل Application Service ها یا Command/Query Handler ها (در صورت استفاده از CQRS)، DTO ها و منطق هماهنگی تراکنش‌ها. این لایه می‌داند «چه کاری باید انجام شود» اما «چگونگی» منطق کسب و کار را به Domain واگذار می‌کند.

لایه	اجزا / مسئولیت‌ها
Domain Layer	قلب سیستم و بدون وابستگی Domain Event, Domain Service, Aggregate, Value Object, Entity
Application Layer	هماهنگی Use Case ها، DTO ها، مدیریت جریان اجرای عملیات
Presentation Layer	Controller ها، API، UI
Infrastructure Layer	پایاده‌سازی Repository، پایگاه‌داده، سرویس‌های خارجی و ابزارهای زیرساختی

ارتباط DDD با سایر الگوهای معماری:

CQRS (Command Query Responsibility Segregation): جداسازی مسیر نوشتن (Command) از مسیر خواندن (Query) است. در سیستم‌های پیچیده DDD، اغلب مدل نوشتن (که از قوانین Aggregate پیروی می‌کند) با مدل خواندن (که برای نمایش بهینه شده) متفاوت است. CQRS معمولاً (نه همیشه) همراه با DDD استفاده می‌شود، اما اجباری نیست! می‌توان DDD را بدون CQRS و برعکس هم پیاده کرد. (در بخش‌های بعدی با CQRS آشنا خواهید شد!)

Event Sourcing: در Event Sourcing، به جای ذخیره وضعیت فعلی یک Aggregate، دنباله‌ای از رویدادها (همان Domain Event ها) ذخیره می‌شود و وضعیت فعلی با «پخش مجدد» (Replay) این رویدادها بازسازی می‌شود. این الگو با DDD هم‌راستایی طبیعی دارد چون هر دو روی رفتار و رویدادهای دامنه تمرکز دارند، اما استفاده از یکی، دیگری را الزامی نمی‌کند.

Microservice ها: Bounded Context در DDD اغلب مبنای طبیعی برای تعیین مرز Microservice ها است. با این حال، DDD الزاماً به معنای میکروسرویس نیست بلکه می‌توان یک Modular Monolith ساخت که در آن هر Bounded Context یک ماژول مجزا (با مرزهای کد واضح) درون یک برنامه واحد است. این رویکرد اغلب برای شروع پروژه‌ها هوشمندانه‌تر است. توضیح بیشتر در مورد Microservice ها

از حوزه این کتاب خارج است! پیشنهاد می شود به آموزش های مختص **Microservice** ها علل الخصوص دوره معماری نرم افزار در **Microservice** ها در آموزشگاه تحلیل داده استاد محمد آزاد مراجعه فرمایید!

 **مثال کامل و پایانی DDD:** این یک مثال کامل از لایه های مختلف یک فروشگاه آنلاین می باشد! در صفحات قبل، یک بار این مثال مطرح شد ولی در نهایت با تفکیک لایه های DDD در آن، بحث DDD را به پایان می رسانیم!

1- مدل دامنه (لایه Domain): در اینجا، Order یک Aggregate Root و OrderLine یک Aggregate داخل آن است!

```
#include <vector>
#include <stdexcept>
#include <algorithm>
// فرض بر این است که این انواع قبلاً تعریف شده اند
// OrderId, CustomerId, ProductId, Money
// OrderLine, IDomainEvent, OrderConfirmedEvent, DomainException
enum class OrderStatus {
    Draft,
    Confirmed
};

class Order {
private:
    OrderId id;
    CustomerId customerId;

    std::vector<OrderLine> lines;

    OrderStatus status = OrderStatus::Draft;

    std::vector<std::shared_ptr<IDomainEvent>> domainEvents;

public:
    Order(
        const OrderId& id,
        const CustomerId& customerId)
        : id(id),
          customerId(customerId) {}

    void addLine(
        const ProductId& productId,
        int quantity,
        const Money& unitPrice) {
        ensureDraft();

        if (quantity <= 0)
            throw DomainException("تعداد باید مثبت باشد");

        lines.emplace_back(
            productId,
            quantity,
            unitPrice
        );
    }

    void confirm() {
        ensureDraft();
        if (lines.empty())
            throw DomainException("سفارش خالی قابل تأیید نیست");
    }
};
```

```

        status = OrderStatus::Confirmed;

        domainEvents.push_back(
            std::make_shared<OrderConfirmedEvent>(id)
        );
    }

private:
    void ensureDraft() const {
        if (status != OrderStatus::Draft){
            throw DomainException(
                "این عملیات فقط در وضعیت پیش‌نویس مجاز است"
            );
        }
    }

public:
    const OrderId& getId() const {
        return id;
    }

    const CustomerId& getCustomerId() const {
        return customerId;
    }

    OrderStatus getStatus() const {
        return status;
    }

    const std::vector<OrderLine>& getLines() const {
        return lines;
    }

    const std::vector<std::shared_ptr<IDomainEvent>>& getDomainEvents() const {
        return domainEvents;
    }

    void clearEvents(){
        domainEvents.clear();
    }
};

```

2- لایه Application:

```

#include <memory>
#include <stdexcept>
// فرض بر این است که این انواع قبلاً تعریف شده‌اند
// IOrderRepository, IEventPublisher,
// ConfirmOrderCommand, Order, IDomainEvent,
// NotFoundException
class ConfirmOrderHandler {
private:
    std::shared_ptr<IOrderRepository> repository;
    std::shared_ptr<IEventPublisher> eventPublisher;
public:
    ConfirmOrderHandler(
        std::shared_ptr<IOrderRepository> repository,
        std::shared_ptr<IEventPublisher> eventPublisher)
        : repository(std::move(repository)),
          eventPublisher(std::move(eventPublisher)) {}
};

```

```

void handle(const ConfirmOrderCommand& command) {
    auto order = repository->getById(command.getOrderId());

    if (!order) {
        throw NotFoundException("سفارش یافت نشد");
    }
    //منطق کسبوکار داخل Aggregate
    order->confirm();

    repository->saveChanges();
    //انتشار Domain Event ها
    for (const auto& domainEvent : order->getDomainEvents()) {
        eventPublisher->publish(domainEvent);
    }
    //پاک کردن Event ها پس از انتشار
    order->clearEvents();
}
};

```

3- لایه Infrastructure (پیاده‌سازی Repository): این لایه همیشه لایه چهارم یعنی لایه بیرونی فرض می‌شود!

```

#include <memory>

class EfOrderRepository : public IOrderRepository {
private:
    AppDbContext& context;

public:
    explicit EfOrderRepository(AppDbContext& context)
        : context(context) {}

    std::shared_ptr<Order> getById(const OrderId& id) override {
        return context.orders().find(id);
    }

    void add(const std::shared_ptr<Order>& order) override {
        context.orders().add(order);
    }

    void saveChanges() override {
        context.saveChanges();
    }
};

```

4- لایه Presentation: به عنوان لایه سوم که واسط کاربری است را با فریم ورک محبوب و سریع drogon

مثال می‌زنیم! Drogon از معماری MVC تبعیت می‌کند و مغز فرمانها توسط Controller با داشتن Model مدیریت می‌شود و View هم از نوتیفیکیشن‌های Model خود را Refresh و Update می‌کند و مستقیماً دستورات کاربر را به Controller خود انتقال می‌دهد!

```

#include <drogon/drogon.h>

class OrdersController {
private:
    std::shared_ptr<ConfirmOrderHandler> handler;

public:
    explicit OrdersController(
        std::shared_ptr<ConfirmOrderHandler> handler)
        : handler(std::move(handler)) {}
};

```

```

    پارامتر دوم کانفرم از شما تابعی را دریافت می کند با ورودی ریسپانس و بدون خروجی
    در اصل پس از دریافت ریسپانس، آنرا به کال بک ورودی ارسال خواهد کرد
    void confirm(
        const drogon::HttpRequestPtr& request,
        std::function<void(const drogon::HttpResponsePtr*)>&& respcallback,
        const std::string& id) {
        try {
            //Guid / UUID به string تبدیل
            Guid guid = Guid::parse(id);

            ConfirmOrderCommand command(
                OrderId(guid)
            );

            handler->handle(command);
            // HTTP 204 No Content
            // عملیات با موفقیت انجام شد؛ ولی محتوایی ندارم که در response body به تو بدهم
            auto response = drogon::HttpResponse::newHttpResponse();
            response->setStatusCode(
                drogon::k204NoContent
            );
            respcallback(response);
        }
        catch (const NotFoundException&) {
            auto response = drogon::HttpResponse::newHttpResponse();

            response->setStatusCode(
                drogon::k404NotFound
            );

            respcallback(response);
        }
        catch (const std::exception& ex) {
            auto response = drogon::HttpResponse::newHttpResponse();

            response->setStatusCode(
                drogon::k500InternalServerError
            );

            respcallback(response);
        }
    }
};

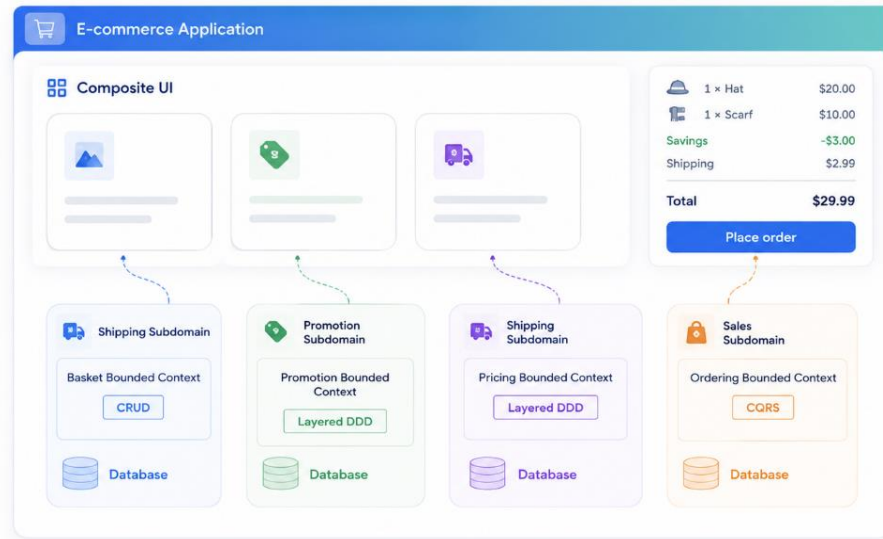
```

Composit UI  به نوعی از UI گفته می شود که از چند سیستم جداگانه ساخته می شود! به بیان بهتر وقتی برای ساخت یک UI نمایشی پیچیده با چندین نوع مختلف موجودیت و داده، مجبوریم از انواع Aggregate ها استفاده کنیم، به آن Composit UI گفته می شود! از طرفی امکان دارد سرور و دیتابیس هر کدام از این سرویس ها، جدای از دیگر سرویس ها باشد! برای مثال گفته می شد صفحه Facebook توسط سرورهای گوناگون ساخته می شود و بخش های مختلف هدر همچون تعداد درخواست های جدید دوستی یا تعداد پیغام های خصوصی جدید و ... به صورت مجزا از چندین سرور به دست شما می رسد! به سیستم UI فیسبوک که اجزای صفحه نمایش را از طریق چندین سرور و سیستم مختلف می سازد، Composit UI گفته می شود!

Composite UI

A composite UI is a user interface made up of loosely coupled components.

A composite UI shows data from multiple bounded contexts on the same web page.



مزایا و معایب DDD 🤖

مزایا ✅

- همسویی بهتر بین کسب و کار و فناوری به واسطه زبان مشترک.
- کد قابل نگهداری تر: چون منطق کسب و کار در یک مکان مشخص (Aggregate ها) متمرکز است، نه پراکنده در Service Layer.
- مرزهای واضح: که مقیاس پذیری تیم و امکان تجزیه به Microservice را تسهیل می کند.
- تست پذیری بالا: چون لایه Domain مستقل از فریم ورک و پایگاه داده است.
- انعطاف در برابر تغییرات کسب و کار: چون مدل دامنه بازتاب واقعیت کسب و کار است، نه ساختار پایگاه داده.

معایب و چالش ها ❌

- پیچیدگی اولیه بالا: برای پروژه های ساده (CRUD ساده)، DDD اغلب بیش از حد پیچیده و غیر ضروری است.
- نیاز به دسترسی مداوم به متخصصان دامنه: بدون همکاری نزدیک با کارشناسان کسب و کار، ساخت زبان مشترک و مدل صحیح دشوار است.
- منحنی یادگیری بالا: تیم های تازه کار در DDD معمولاً الگوها را نادرست پیاده سازی می کنند (مثلاً Anemic Domain Model که ظاهراً DDD است اما در واقع منطق در آن جایی ندارد).
- زمان توسعه اولیه بیشتر: طراحی صحیح Aggregate ها و Bounded Context ها زمان بر است.



چه زمانی از DDD استفاده کنیم؟ DDD برای دامنه‌های پیچیده با منطق کسب و کاری غنی (Core Domain) بسیار مناسب است. مثل سیستم‌های بانکی، بیمه، لجستیک پیچیده، یا پلتفرم‌های چندوجهی. اما در مورد سیستم‌های ساده اداری، ابزارهای داخلی ساده، یا MVP اولیه، معمولاً معماری‌های ساده‌تر (مثل Transaction Script یا CRUD مستقیم) کارآمدتر و سریع‌تر هستند. مارتین فاولر این را با اصطلاح «پیچیدگی تصادفی در برابر پیچیدگی ذاتی» توضیح می‌دهد: DDD باید فقط جایی به کار رود که پیچیدگی از دل خود کسب‌وکار می‌آید.

❌ نامناسب برای DDD	✅ مناسب برای DDD
اپلیکیشن‌های CRUD ساده	منطق کسب‌وکار پیچیده
ابزارهای داخلی کوچک	دامنه در حال تغییر مداوم
پروژه‌های با عمر کوتاه	تیم‌های بزرگ با چند Bounded Context
زمانی که سرعت توسعه اولویت اصلی است	نیاز به همکاری نزدیک با Domain Experts

❌ اشتباهات رایج در پیاده‌سازی DDD

1- Anemic Domain Model: ساخت Entityهایی که فقط Getter/Setter دارند و تمام منطق در Service Layer است این دقیقاً نقطه مقابل هدف DDD است.

2- Aggregateهای بیش از حد بزرگ: قرار دادن بسیاری از موجودیت‌ها در یک Aggregate باعث قفل‌شدن تراکنشی و افت کارایی می‌شود.

3- نادیده گرفتن Bounded Context: تلاش برای ساخت یک «مدل جهانی» واحد برای کل سازمان که همه‌جا صدق کند (این عملاً غیرممکن و ضدالگوی DDD است).

4- افراط در استفاده از الگوها: استفاده از CQRS، Event Sourcing و Microservice در پروژه‌ای که نیازی به این پیچیدگی ندارد.

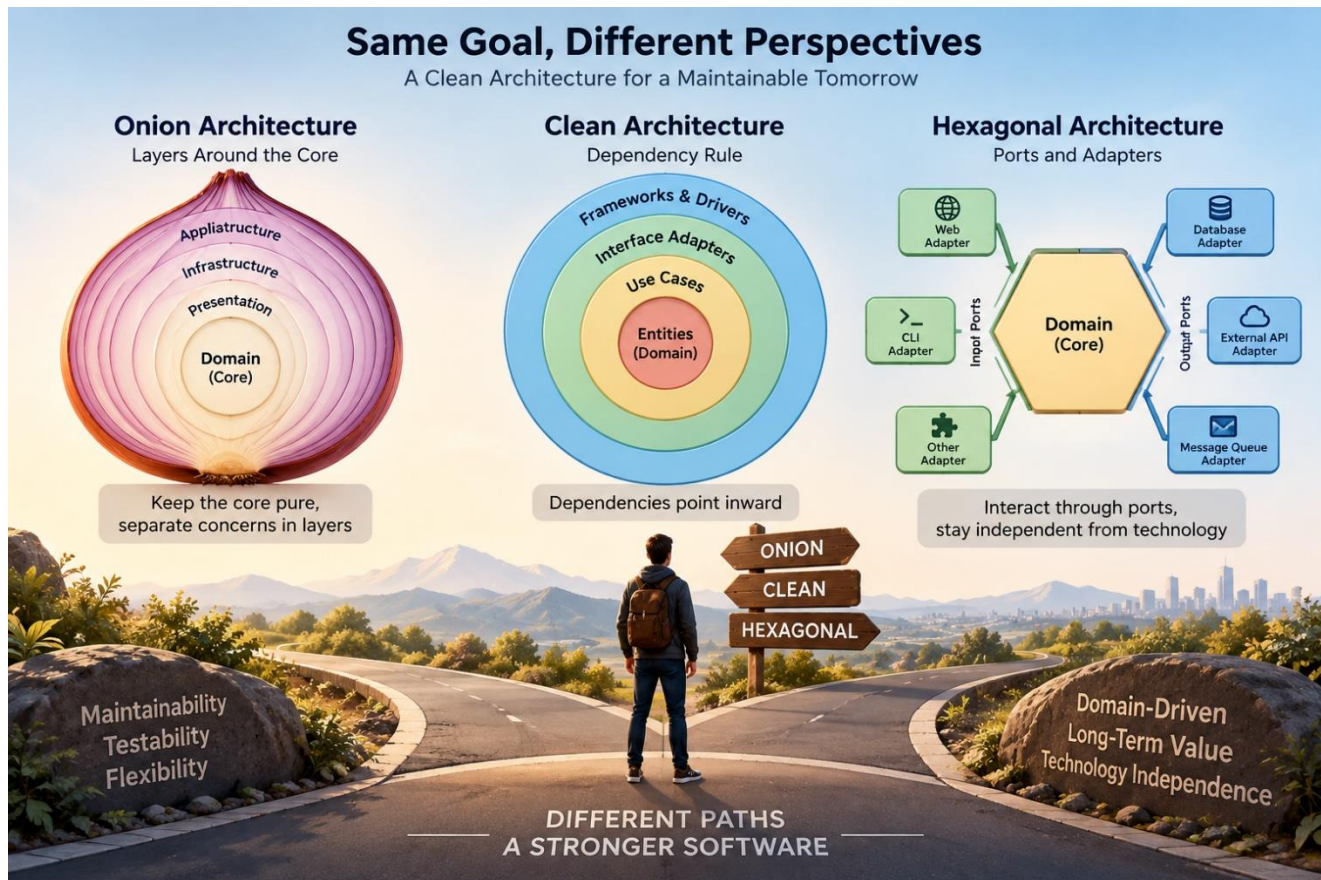
5- نادیده گرفتن زبان مشترک: نوشتن کد با اصطلاحات فنی به جای اصطلاحات واقعی کسب و کار.

📌 در بخش نهایی و برای تکمیل توضیحات بخش DDD به طور خلاصه:

به قول دوست عزیز، استاد آزاد، DDD یعنی تفرقه بینداز و حکومت کن! به بیان بهتر یعنی یک چیز بزرگ را بشکان و آنرا حل کن! یک فیل را چگونه می‌خورند؟ لقمه لقمه!

DDD تلاش می‌کند Domain‌های پروژه را از هم تفکیک کند. Domain مهمترین بخش و به بیان بهتر قلب هر نرم افزار و بخش اصلی کسب و کار است که کسب و کار، به آن شهره است!

- معماری های لایه ای (Clean, Onion, Hexagonal (Port & Adapter): هر سه از معروف ترین الگوهای معماری برای طراحی و ساختاردهی نرم افزارهای سمت Server، Backend، API و Microservices هستند.



همان طور که MVC، MVP و MVVM را می توان از الگوهای معروف برای ساختاردهی اپلیکیشن های Presentation در نظر گرفت، Clean، Onion و Hexagonal نیز از الگوهای بسیار معروف برای ساختاردهی لایه های داخلی و وابستگی های نرم افزار در Backend محسوب می شوند.

یکی از اصول فکری مهمی که باعث زایش انواع معماری های نرم افزار نوین شد، لایه بندی و جداسازی منطق از پیاده سازی، جدا کردن مسئولیت ها به بخش های جداگانه جهت سادگی عیب یابی و مدیریت عملکرد ها بوده است! یکی از رایج ترین الگوهای لایه ای، معماری لایه ای سنتی (Layered / N-Tier) بود که در آن، جهت وابستگی رو به پایین بود:

Presentation → Business Logic → Data Access → Database

✗ **مشکل این الگو:** منطق کسب و کار مستقیماً به جزئیات دیتابیس و فریم ورک وابسته می شد، به بیان بهتر جهت وابستگی رو به پایین بود؛ یعنی لایه Business Logic مستقیماً به لایه Data Access (و از آن طریق، به ORM و پایگاه داده) وابسته می شد. این یعنی:

تست کردن منطق کسب و کار بدون پایگاه داده واقعی دشوار بود. یعنی نمی توانستیم با جایگزین کردن دیتابیس، روی دیتابیس های Mock و Fake تست انجام دهیم! تغییر فناوری پایگاه داده یا فریم ورک وب، منطق کسب و کار را هم تحت تأثیر قرار می داد (منطق شکننده).

قلب سیستم (منطق کسب و کار) به جای اینکه مستقل باشد، اسیر جزئیات فنی بود.

هر سه معماری Clean، Onion و Hexagonal برای حل (دقیقاً) همین مشکل به وجود آمدند، با یک اصل مشترک به نام "اصل وارونگی وابستگی" یا "Dependency Inversion Principle - DIP"

پس هر سه معماری موضوع این بخش از کتاب، برای حل این مشکلات به وجود آمدند! در هر 3 معماری مذکور، جهت وابستگی همیشه به سمت مرکز/داخل است، نه به سمت جزئیات فنی! کد پیرامونی (وب، پایگاه داده، فریم ورک) می تواند به هسته وابسته باشد، اما هسته هرگز نباید بداند کد پیرامونی چیست یا چگونه کار می کند.

معماری	سال معرفی	معرف	تمرکز اصلی
Hexagonal	2005	Alistair Cockburn	تقارن Port ها و Adapter ها
Onion	2008	Jeffrey Palermo	جایگاه مرکزی Domain Model (ریشه در معماری DDD)
Clean	2012	Robert C. Martin	تدوین یک قانون و زبان عمومی برای کل خانواده

 **نکته مهم:** Onion قبل از Clean معرفی شد ولی Clean همان Onion تکامل یافته است و هر دو دقیقاً یک ساختار با نام گذاری مختلف در لایه ها است!

 چرا این سه معماری شبیه هم به نظر می رسند؟

 **اصل بنیادین مشترک: قانون وابستگی (Dependency Rule)**

پایه هر سه معماری، اصل وارونگی وابستگی (Dependency Inversion Principle – DIP) است: ماژول های سطح بالا (منطق کسب و کار) نباید به ماژول های سطح پایین (جزئیات فنی) وابسته باشند؛ هر دو باید به انتزاعات (Interface) وابسته باشند و نتیجه این اصل در هر سه معماری یکسان است: جهت وابستگی همیشه به سمت مرکز/هسته است! نه به سمت جزئیات فنی! کد پیرامونی (وب، دیتابیس، فریم ورک) می تواند به هسته وابسته باشد، اما هسته هرگز نباید بداند کد پیرامونی چیست یا چگونه کار می کند! (توضیحات DIP در قسمت Solid توضیح داده شد و همچنین در قسمت معرفی الگوی طراحی Dependency Injection هم گریزی به معرفی این اصل جادویی داشتیم!)

```
class OrderService {
public:
    void createOrder(const Order& order) {
        mysql.query("INSERT INTO orders ..."); // مستقیم به MySQL ارسال می کند
        smtp.send("order@shop.com", ...); // مستقیم Email می فرستد
        http.post("https://payment.api/..."); // مستقیم HTTP می زند
    }
};
```

در کلاس فوق، مشکلاتی رخ می دهد!

- تست بدون MySQL واقعی قابل اجرا نیست! فرض کنید می خواهید منطق کسب و کار را تست کنید ولی با ایراد در ارتباط با DataBase تست شما به مشکل می خورد و عملاً تست منطق کسب و کار با اختلال در دیتابیس به مشکل می خورد!
- تغییر از دیتابیس MySQL به PostgreSQL یعنی تغییر اصل و کل کد!
- تعویض سرویس ایمیل به سیستم دیگری یعنی تغییر کل کد (تست بدون داشتن ارتباط اینترنتی ممکن نیست!) در طول عمر یک پروژه، فناوری های زیرساختی معمولاً عوض می شوند! امروز درگاه داخلی فردا درگاه خارجی، امروز دیتابیس PostgreSQL فردا MySQL و ...

در طراحی سنتی، هر کدام از این تعویض ها یعنی بازنویسی بخشی از منطق کسب و کار! دقیقاً همان چیزی که Hexagonal با انتزاع Port از آن جلوگیری می کند.

🎯 هدف ما از استفاده از این هر یک از این 3 معماری:

- 1- مستقل از فریم ورک باشد.
- 2- قابل تست است (بدون UI، پایگاه داده، وب سرور یا هر عنصر خارجی).
- 3- مستقل از UI است. به بیان بهتر یکی از سم های مهلک تست، وابسته بودن یک عملیات به کلیک روی دکمه یا درگ کردن یک SliderBar و ... است! وقتی شما می خواهید با اجرای توابع تست پشت سر هم از یک جا، عملکردهای یک سیستم را تست کنید، باید نرم افزاری تولید کرده باشید که عملکردهایش مستقل از UI باشد! از طرفی استقلال از UI یعنی اگر در آینده بخواهید مدل UI را تغییر دهید، نباید اصل پروژه تان تغییر کند! (مثال نمایش ساعت دیجیتال یا عقربه دار در مثال MVC و MVP و MVVM)
- 4- مستقل از پایگاه داده است. برای تغییر پایگاه داده کافیت Repository اتصال به آنرا تغییر دهید!
- 5- مستقل از هر عامل یا سیستم خارجی است.

🏠 Hexagonal Architecture (Ports & Adapters)

این معماری در سال ۲۰۰۵ توسط آلیستر کاکبرن (Alistair Cockburn) معرفی شد. انگیزه او این بود: «چرا باید فرقی کند که یک برنامه از طریق رابط کاربری وب، یک تست خودکار، یا یک اسکریپت خط فرمان استفاده شود؟ او می خواست برنامه ای بسازد که بتوان آن را بدون هیچ رابط کاربری خاصی اجرا و تست کرد!

🏠 لایه ها و مفاهیم کلیدی Port و Adapter:

۱. هسته برنامه (Application Core): شامل منطق و قوانین کسب و کار، Use Case ها، Application/Domain Service ها و Exception های مرتبط با منطق برنامه است. کاملاً مستقل از فناوری ها و جزئیات بیرونی مانند Database، Web API و Framework ها است، در مرکز معماری قرار دارد و با دنیای بیرون از طریق Port ها ارتباط برقرار می کند.
 ۲. Port: یک Interface (قرارداد) است (پایه سازی ندارد و فقط نگاشت یا شابلون درخواست و دریافت ها را مشخص می کند) که هسته تعریف می کند و تعیین می کند «چه چیزی نیاز دارم» یا «چه چیزی می دهم»! port دو حالت دارد:
 - **Inbound / Driving Port**: راهی که دنیای بیرون از طریق آن به هسته دستور می دهد (صدا زدن Use Case ها). در واقع این نوع Port خود Use Case نیست؛ بلکه Interface/Contract ای است که Use Case را Expose می کند. مثلاً `CreateOrderService` یک Inbound Port است و `CreateOrderService` پیاده سازی آن در هسته! پس این نوع Port قراردادی است که هسته برای دریافت درخواست از دنیای بیرون تعریف می کند و معمولاً نمایانگر یک Use Case است.
 - **Outbound / Driven Port**: راهی که هسته از طریق آن برای اجرای درخواست در سرویس های خارج از API (استفاده از ابزارهای لایه Infrastructure مثل ذخیره ساز/دیتابیس، ارسال SMS یا Email) با دنیای بیرون ارتباط می گیرد. مثلاً `IOrderRepository` یا `ISmsSender` از نوع Outbound Port هستند.
- سناریوی نمونه:** کاربر با فراخوانی Route با عنوان `create_order` دستور ساخت Order را صادر می کند. این Route توسط یک Controller (از نوع Primary Adapter) به Inbound Port (مانند `CreateOrderUseCase`) متصل شده و دستور را به هسته منتقل می کند. پیاده سازی این Port (یعنی `CreateOrderService`) بر اساس اعتبارسنجی های لازم و در

صورت تایید، برای ذخیره سازی به یک Outbound Port (مانند IOrderRepository) دستور ساخت Order جدید را می دهد و نهایتاً Repository مناسب (از نوع Secondary Adapter)، آن را در دیتابیس می سازد!

۳. **Adapter**: پیاده سازی واقعی یک Port است که آن Interface را به یک تکنولوژی یا دنیای بیرون وصل می کند و دو نوع دارد:

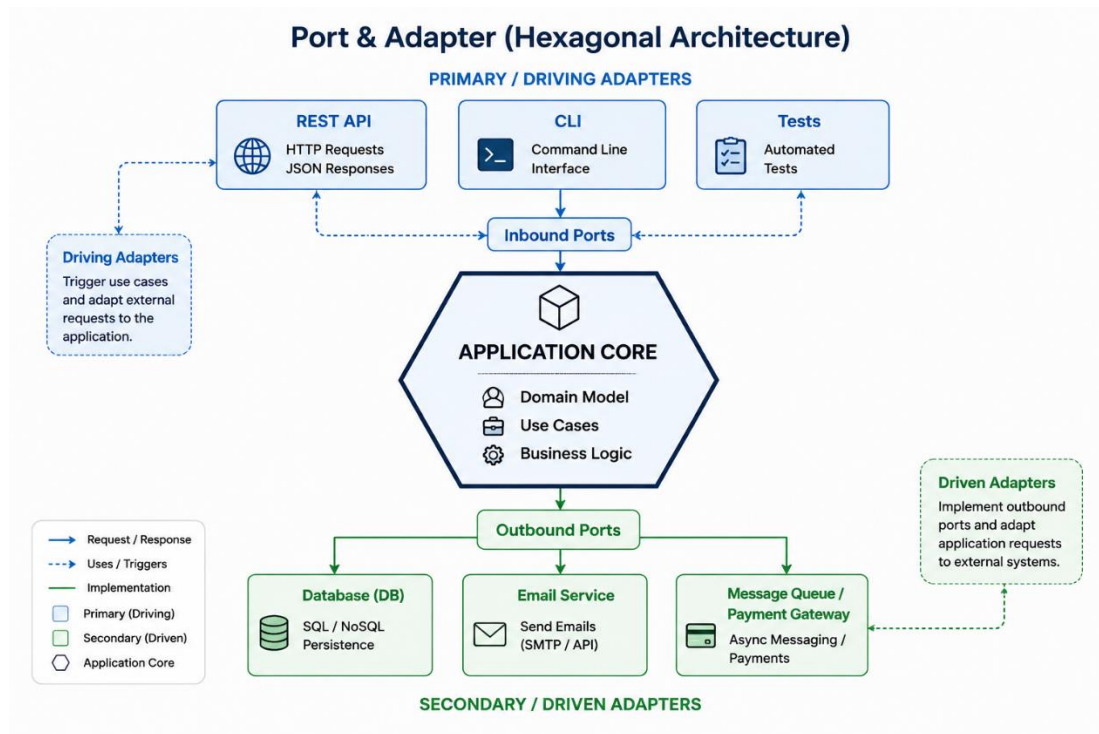
- **Primary / Driving Adapter**: چیزی که هسته را فرا می خواند (رابط وب، CLI، یا تست). به بیان بهتر، کاربران بر اساس نیازهایشان درخواستشان را سمت آن ها هدایت می کنند (مثل درخواست ساخت یا ویرایش Order). پس این نوع Adapter چیزی است که از بیرون، هسته و Use Case های آن را فراخوانی می کند؛ مانند REST Controller، CLI یا تست ها. این Adapter درخواست ورودی را دریافت کرده و آن را به Inbound Port مناسب هدایت می کند.

- **Secondary / Driven Adapter**: چیزی که هسته آن را فرا می خواند. این Adapter ها Port های خروجی (Outbound Ports) را پیاده سازی می کنند تا با لایه های بیرونی در لایه چهارم یا Infrastructure مثل دیتابیس، فریم ورک های خارجی و API های دیگر تعامل داشته باشند؛ مانند Repository دیتابیس، Payment Gateway و Email Sender.

(نکته: معماری Hexagonal اصلاً درباره تعداد لایه ها صحبت نمی کند و فقط یک مرز دوگانه دارد: هسته در برابر همه چیز دیگر. این معماری روی تقارن Adapter ها و تست پذیری کامل تمرکز دارد.)

💡 چرا نام شش ضلعی؟

کاکبرن عمداً شکل شش ضلعی را انتخاب کرد تا از تصویر ذهنی «بالا / پایین» در نمودارهای لایه ای سنتی فرار کند. در این دید، همه Adapter ها از نظر مفهومی هم ارز هستند؛ تمایز اصلی بر اساس جهت جریان کنترل (Driving در برابر Driven) است، نه نوع فناوری.



✿ پیاده سازی کامل و واقعی با ++C :

برای ملموس تر شدن موضوع، یک سیستم واقعی تر (فروشگاه آنلاین) را با اصطلاحات Hexagonal پیاده می کنیم که شامل چند Port، چند Adapter، Composition Root و تست است! دقیقاً همان روندی که در پروژه های واقعی طی می شود. در این کد، هیچ اشاره ای به MySQL، SMTP یا Stripe در هسته برنامه وجود ندارد.

هسته Core بدون الزام به لایه بندی داخلی مشخص، در مرکز قرار دارد و Port ها و Adapter ها آن را احاطه کرده اند.

به صورت ساده و خلاصه:

کاربر درخواست خود را به Route ها می فرستد! Route ها و Controller ها در واقع Inbound Adapter هستند؛ آن ها درخواست ورودی کاربر را در قالب HTTP/JSON یا CLI دریافت می کنند و آن را به فرمت قابل فهم برای هسته (Core) تبدیل می کنند و سپس Inbound Port که یک Interface تعریف شده در Core است را فراخوانی می کنند.

درخواست با فرمت استاندارد وارد Core می شود و نوبت اجرای منطق کسب و کار (Domain) است! Domain می گوید: "من می خواهم یک Order جدید را ذخیره کنم!" برای Domain مهم نیست داده کجا ذخیره می شود؛ این قاعده و دستور حفظ داده توسط Outbound Port و در اصل توسط یک Interface مثل OrderRepository تعریف می شود. حالا ما می توانیم با انواع Secondary Adapter ها (در قالب انواع Repository ها) این کار را انجام دهیم:

- یک Adapter می تواند داده را در PostgreSQL ذخیره کند،

- دیگری در MySQL،

- و دیگری روی حافظه (In-Memory) یا فایل JSON!

نحوه ذخیره سازی، در پیاده سازی هر کدام از این Adapter ها تعریف می شود، اما برای Domain، قاعده و نحوه دستور دادن یکی است و آن را Outbound Port مشخص می کند.

پس از انجام ذخیره سازی، نتیجه عملیات از Secondary Adapter به Domain برمی گردد. سپس Domain کارش را تمام کرده و نتیجه را به Inbound Adapter همان Controller ریترن (Return) می کند. در نهایت، Inbound Adapter نتیجه را به فرمت مناسب (مثلاً پاسخ HTTP) به کلاینت برمی گرداند.

تشبیه پریز و آداپتور در معماری شش ضلعی (Hexagonal Architecture)

- **Domain**: در این تشبیه، دستگاه الکترونیکی حکم هسته و دامنه (Application Core / Domain) را دارد. دستگاه منطق و نحوه کار خود را دارد و فقط می گوید: «برای روشن شدن و کار کردن، به برق ۱۲ ولت مستقیم (DC) نیاز دارم.»
- **Inbound Port (پریز ورودی دستگاه)**: درگاه یا ورودی استاندارد روی بدنه دستگاه است. دستگاه مشخص می کند که برق ورودی باید دقیقاً چه ویژگی هایی داشته باشد تا کار کند.

- **Inbound Adapter (آداپتور برق)**: ابزاری است که منبع انرژی بیرونی (مثلاً برق ۲۲۰ ولت شهری، پنل خورشیدی، یا باتری ماشین) را می گیرد و آن را به برق ۱۲ ولت استاندارد مورد نیاز Inbound Port تبدیل می کند. برای دستگاه مهم نیست منبع اولیه برق چیست (برق شهری، ژنراتور، باتری خورشیدی، پاوربانک)؛ آداپتور وظیفه دارد آن را به زبان قابل فهم دستگاه تبدیل کند.

- **Outbound Port (درگاه خروجی دستگاه به وسایل جانبی):** استاندارد است که دستگاه برای استفاده از امکانات بیرونی

تعریف می‌کند؛ مثلاً یک پورت استاندارد صدا که می‌گوید: «من سیگنال صوتی خام تولید می‌کنم و به دستگاه جانبی همچون Speaker می‌دهم.»

- **Outbound / Secondary Adapter (اسپیکر یا هدفون):** پیاده‌سازی واقعی وسیله جانبی است. این وسیله می‌تواند یک

هدفون ارزان‌قیمت، یک اسپیکر بلوتوثی یا یک سیستم صوتی پیشرفته باشد. تا زمانی که این وسایل با Outbound Port دستگاه سازگار باشند، دستگاه بدون نیاز به هیچ تغییری در درون خود، صدا را پخش خواهد کرد.

🏠 ساختار فایل‌های پروژه:

```
project-root/
├── src/
│   ├── domain/
│   │   ├── entities/
│   │   │   └── Order.hpp
│   │   ├── exceptions/
│   │   │   └── product_exceptions.hpp
│   │   └── services/
│   │       └── PricingService.hpp
│   ├── ports/
│   │   ├── inbound/
│   │   │   ├── ICreateOrderUseCase.hpp
│   │   │   ├── IGetOrderUseCase.hpp
│   │   │   └── ICancelOrderUseCase.hpp
│   │   └── outbound/
│   │       └── OutboundPorts.hpp
│   ├── application/
│   │   └── use-cases/
│   │       ├── CreateOrderUseCase.hpp
│   │       └── CreateOrderUseCase.cpp
│   ├── adapters/
│   │   └── secondary/
│   │       ├── InMemoryProductRepository.hpp
│   │       ├── InMemoryOrderRepository.hpp
│   │       ├── SimpleIdGenerator.hpp
│   │       ├── MockPaymentGateway.hpp
│   │       └── ConsoleEmailSender.hpp
│   └── config/
│       └── AppContainer.hpp
└── main.cpp
```

```
// src/domain/exceptions/product_exceptions.hpp
#pragma once

#include <exception>
#include <string>
#include <utility>
// _____
// Domain Exceptions
// _____

struct DomainException : std::exception {
    std::string msg;

    explicit DomainException(std::string message)
        : msg(std::move(message)) {}

    const char* what() const noexcept override {
        return msg.c_str();
    }
};

struct InsufficientStockException : DomainException {
    InsufficientStockException(const std::string& productId, int requested, int available)
        : DomainException("Product " + productId +
            ": requested " + std::to_string(requested) +
            " but only " + std::to_string(available) + " available") {}
};

struct ProductNotFoundException : DomainException {
    explicit ProductNotFoundException(const std::string& id)
        : DomainException("Product not found: " + id) {}
};

// src/domain/entities/Order.hpp
#pragma once

#include <string>
#include <vector>

#include "src/domain/exceptions/product_exceptions.hpp"
// _____
// Product: مثلا آیفون 17 نقره ای فلان و بهمان
// _____
struct Product {
    std::string id;
    std::string name;
    double price;
    int stock; // تعداد موجودی کالا در انبار
};
// _____
// Order Item: یک آیتم خرید در لیست مثلا در لیست خرید 1 عدد آیفون 17 نقره ای با
// قیمت فلان خرید زده اید
// _____
struct OrderItem {
    std::string productId;
    int quantity; // تعداد کالا در سفارش مشتری
    double unitPrice;

    /* قیمت واحد، قیمت لحظه خرید توسط کاربر است که شاید با قیمت کنونی فرق داشته باشد.
    قیمت Product، قیمت به روز فعلی محصول است؛ اما unitPrice قیمت لحظه خرید توسط خریدار
    است. فرض کنید پس از خرید توسط مشتری، قیمت کنونی محصول گران تر یا ارزان تر شود و

```

هنوز Order به جهت مشکلات دیتابیس یا هر نوع تأخیر ثبت نشده باشد. در این حالت باید
قیمت زمان خرید در Order ثبت شود/

```
double total() const {
    return unitPrice * quantity;
}

};

// -----
// Order: یک خرید کامل یا فاکتور و لیست اقلام و آی دی مشتری
// -----
struct Order {
    std::string id; // آی دی فاکتور خرید
    std::string customerId; // آی دی خریدار
    std::vector<OrderItem> items; // لیست خرید

    enum class Status {
        Pending,
        Confirmed,
        Shipped,
        Cancelled
    };

    Status status = Status::Pending;

    double totalAmount() const {
        double total = 0;
        for (const auto& item : items) {
            total += item.total();
        }
        return total;
    }

    // یک متد ساده برای نشان دادن منطق دامنه
    void addItem(const OrderItem& item, int currentStock) {
        if (status != Status::Pending) {
            throw DomainException(
                "Cannot add items to a non-pending order"
            );
        }
        if (item.quantity > currentStock) {
            throw InsufficientStockException(
                item.productId,
                item.quantity,
                currentStock
            );
        }
        items.push_back(item);
    }
};
```

سرویس های Domain (Domain Services): 

```
// src/domain/services/PricingService.hpp
#pragma once
#include <vector>
#include "src/domain/entities/Order.hpp"

class PricingService {
public:
    static double calculateTotal(
        const std::vector<OrderItem>& items
    );
};
```

```

static double applyDiscount(
    double amount,
    double discountPercent
);

static double calculateTax(
    double amount,
    double taxRate
);
};

```

Inbound Ports (Driving Ports): اینترفیس‌هایی که لایه **Application** تعریف می‌کند تا بگوید "سیستم



من چه کارهایی می‌تواند انجام دهد!": این Interface ها تعریف می‌کنند کاربران، چه درخواست‌هایی را با چه فرمتی می‌توانند

ارسال کنند و پاسخ را در چه فرمتی دریافت کنند! (مثل ساختن سفارش جدید)

```

//src/ports/inbound/ICreateOrderUseCase.hpp
#pragma once

#include <string>
#include <vector>
#include "src/domain/entities/Order.hpp"
namespace Ports::Inbound {

struct CreateOrderRequest { // ساختار درخواست ساخت یک اوردر
    std::string customerId;
    std::vector<OrderItem> items;
};

struct CreateOrderResponse { // ساختار پاسخ و نتیجه درخواستی که بالاتر تعریف کردیم
    bool success;
    std::string orderId;
    std::string message;
    double totalAmount;
};

class ICreateOrderUseCase { // پورت این باند اصلی یوزکیس ساخت یک اوردر جدید
public:
    virtual ~ICreateOrderUseCase() = default;
    // دستور ساخت یک اوردر با ورودی ریکوئست و خروجی ریسپانس تعریف شده بالا
    virtual CreateOrderResponse execute(const CreateOrderRequest& request) = 0;
};

} // namespace Ports::Inbound
//src/ports/inbound/IGetOrderUseCase.hpp
#pragma once
#include <optional>
#include <string>
#include "src/domain/entities/Order.hpp"

namespace Ports::Inbound {
// یک این باند پورت ساده ای که نیاز به تعریف پکیج خاص درخواست و نتیجه ندارد
class IGetOrderUseCase {
public:
    virtual ~IGetOrderUseCase() = default;
    // بازگرداندن optional نشان می‌دهد که ممکن است سفارش پیدا نشود
    virtual std::optional<Order> execute( const std::string& orderId ) = 0;
};

} // namespace Ports::Inbound

```

```
//src/ports/inbound/ICancelOrderUseCase.hpp
// این باند پورت سوم ساده که نیازی به پکیج اختصاصی درخواست و نتیجه درخواست ندارد
#pragma once
#include <string>
namespace Ports::Inbound {

class ICancelOrderUseCase {
public:
    virtual ~ICancelOrderUseCase() = default;
    // در صورت موفقیت true و در صورت خطا
    // مثلاً سفارش قبلاً کنسل شده false یا Exception برمی‌گرداند
    virtual bool execute(
        const std::string& orderId
    ) = 0;
};

} // namespace Ports::Inbound
```

Outbound Ports (Driven Ports)  اینترفیس‌هایی (مثل Repository ها) که توسط لایه Application تعریف می‌شوند تا نحوه تعامل با لایه ها و دنیای بیرون (دیتابیس، ایمیل و...) را مشخص کنند.

```
// src/ports/outbound/OutboundPorts.hpp
#pragma once

#include <optional>
#include <string>
#include "src/domain/entities/Order.hpp"

namespace Ports::Outbound {
// -----
// Product Repository
// -----
class IProductRepository {
public:
    virtual ~IProductRepository() = default;

    virtual std::optional<Product> findById(const std::string& id) const = 0;

    virtual void save(const Product& product) = 0;

    virtual void update(const Product& product) = 0;
};
// -----
// Order Repository
// -----
class IOrderRepository {
public:
    virtual ~IOrderRepository() = default;

    virtual std::optional<Order> findById(const std::string& id) const = 0;

    virtual void save(const Order& order) = 0;

    virtual void update(const Order& order) = 0;
};
};
```



```

// -----
// Email Sender
// -----
class IEmailSender {
public:
    virtual ~IEmailSender() = default;

    virtual void sendOrderConfirmation(
        const std::string& to,
        const Order& order
    ) = 0;
};

// -----
// Payment Gateway
// -----
class IPaymentGateway {
public:
    struct PaymentResult {
        bool success;
        std::string transactionId;
        std::string errorMessage;
    };
    virtual ~IPaymentGateway() = default;

    virtual PaymentResult charge(
        const std::string& customerId,
        double amount
    ) = 0;
};

// -----
// ID Generator
// -----
class IIdGenerator {
public:
    virtual ~IIdGenerator() = default;

    virtual std::string generate() = 0;
};

} // namespace Ports::Outbound

```

 **پایاده سازی لایه Usecase در Api و منطق آن با پیاده سازی از روی Inbound Ports (اینترفیس):** به سادگی کلاس هایی را در این لایه می نویسیم که اینترفیس های Inbound Ports را پیاده سازی می کند! به بیان بهتر در این لایه، اینترفیس های تعریف شده در Inbound Ports پیاده سازی می شوند. کلاس CreateOrderUseCase وظیفه هماهنگ سازی (Orchestration) بین موجودیت های Domain و سرویس های خارجی را بر عهده دارد.

این کلاس از طریق تزریق وابستگی (Dependency Injection) در Constructor، به پورت های خروجی (Outbound Ports) مانند Repository ها و Payment Gateway وابسته است، اما هیچ اطلاعی از پیاده سازی واقعی آن ها (مثلاً MySQL یا Stripe) ندارد. این موضوع باعث می شود این لایه کاملاً مستقل از فریم ورک و به شدت تست پذیر (Testable) باشد.

```

// src/application/use-cases/CreateOrderUseCase.hpp
#pragma once
#include "src/ports/inbound/ICreateOrderUseCase.hpp"
#include "src/ports/outbound/OutboundPorts.hpp"
namespace Application::UseCases {

```

```

class CreateOrderUseCase : public Ports::Inbound::ICreateOrderUseCase {
private:
    // استفاده از (&) Reference عالی است، زیرا تضمین می‌کند
    // شیئی که ما به آن وابسته هستیم هرگز null نیستند
    Ports::Outbound::IProductRepository& productRepo_;
    Ports::Outbound::IOrderRepository& orderRepo_;
    Ports::Outbound::IEmailSender& emailSender_;
    Ports::Outbound::IPaymentGateway& payment_;
    Ports::Outbound::IIIdGenerator& idGen_;

public:
    CreateOrderUseCase (
        Ports::Outbound::IProductRepository& p,
        Ports::Outbound::IOrderRepository& o,
        Ports::Outbound::IEmailSender& e,
        Ports::Outbound::IPaymentGateway& pay,
        Ports::Outbound::IIIdGenerator& id
    )
        : productRepo_(p),
          orderRepo_(o),
          emailSender_(e),
          payment_(pay),
          idGen_(id) {}

    Ports::Inbound::CreateOrderResponse execute(
        const Ports::Inbound::CreateOrderRequest& request) override;
};

} // namespace Application::UseCases
//src/application/use-cases/CreateOrderUseCase.cpp
#include "src/application/use-cases/CreateOrderUseCase.hpp"

#include "src/domain/entities/Order.hpp"
#include "src/domain/exceptions/product_exceptions.hpp"

namespace Application::UseCases {

Ports::Inbound::CreateOrderResponse
CreateOrderUseCase::execute( const Ports::Inbound::CreateOrderRequest& request) {
    // ساخت شیء Order در حافظه
    // هنوز در دیتابیس ذخیره نشده
    Order order;

    order.id = idGen_.generate();
    order.customerId = request.customerId;
    order.status = Order::Status::Pending;
    // بررسی موجودی و ساخت آیتم‌ها
    // (بدون تغییر در دیتابیس)
    for (const auto& reqItem : request.items) {
        auto productOpt = productRepo_.findById(reqItem.productId);
        if (!productOpt) {
            throw ProductNotFoundException( reqItem.productId );
        }
        Product& product = *productOpt;
        // استفاده از رفرنس برای قابلیت تغییر
        if (product.stock < reqItem.quantity) {
            throw InsufficientStockException(product.id, reqItem.quantity, product.stock);
        }
    }
}

```

```

        order.items.push_back({
            reqItem.productId,
            reqItem.quantity,
            product.price
        });
    }
    // انجام پرداخت
    // (قبل از هرگونه تغییر در دیتابیس)
    auto payResult =
        payment_.charge(
            request.customerId,
            order.totalAmount()
        );
    if (!payResult.success) {
        // پرداخت ناموفق بود، هیچ تغییری در دیتابیس
        // ایجاد نشده است
        return {
            false,
            "",
            "Payment failed: " + payResult.errorMessage,
            0.0
        };
    }
    // پرداخت موفق بود
    // اکنون تغییرات در دیتابیس اعمال می‌شوند
    order.status = Order::Status::Confirmed;
    orderRepo_.save(order);
    // به روزرسانی موجودی محصولات می‌تواند
    // در یک Transaction دیتابیس باشد
    for (const auto& orderItem : order.items) {
        auto productOpt =
            productRepo_.findById(orderItem.productId);
        if (productOpt) {
            productOpt->stock -= orderItem.quantity;
            productRepo_.update(*productOpt);
        }
    }
    // ارسال ایمیل تأیید
    // نکته: در واقعیت ایمیل باید از روی User گرفته شود.
    // اینجا برای سادگی فرض شده customerId همان ایمیل است.
    emailSender_.sendOrderConfirmation(
        request.customerId,
        order
    );
    // بازگرداندن پاسخ موفق
    return {
        true,
        order.id,
        "Order created successfully",
        order.totalAmount()
    };
}

} // namespace Application::UseCases

```

نکته کلیدی: کلاس CreateOrderUseCase هیچ وابستگی مستقیمی به MySQL، SMTP، یا Stripe یا هیچ فناوری خاصی ندارد و تنها به abstractionها یا به عبارتی Portها وابسته است. زیبایی کار کجاست؟ اگر زیرساخت هر کدام از این سرویس‌ها، در لایه چهارم عوض شد، نیازی به اصلاح یا تغییر این لایه‌ها نیست!

 **پیاده سازی لایه Secondary Adapters (آداپترهای ثانویه) یا لایه زیرساخت:** در این قسمت ما پیاده سازی واقعی لایه outbound ports را انجام می دهیم! این لایه همان جایی است که دنیای واقعی (دیتابیس، فایل، سرویس خارجی) به معماری ما متصل می شود. وظیفه این لایه، ترجمه دستورات انتزاعی لایه Application به عملیات واقعی روی سیستم های خارجی (مانند دیتابیس، فایل سیستم یا API های خارجی) است. برای سادگی کار ما در این قسمت، پیاده سازی In-Memory (حافظه رم) را برای ریپازیتوری ها می نویسم. این بهترین انتخاب برای مثال ها و همچنین برای Unit Testing است. مزیت بزرگ این طراحی این است که لایه Application (Use Case ها) هیچ اطلاعی از این پیاده سازی ندارد. در آینده، برای مهاجرت به MySQL یا PostgreSQL، فقط کافی است یک کلاس جدید (مثلاً MySqlOrderRepository) بسازیم که همان اینترفیس را پیاده سازی کند! آن هم بدون اینکه حتی یک خط از کد Use Case تغییر کند! در کد زیر، چندین آداپتور ثانویه را در یک بلوک کد قرار داده ایم!

```

// src/adapters/secondary/InMemoryProductRepository.hpp
#pragma once
#include <optional>
#include <stdexcept>
#include <string>
#include <unordered_map>

#include "src/domain/entities/Order.hpp"
#include "src/ports/outbound/OutboundPorts.hpp"
namespace Adapters::Secondary {

class InMemoryProductRepository
    : public Ports::Outbound::IProductRepository {
private:
    // شبیه‌سازی دیتابیس با استفاده از یک Map
    std::unordered_map<std::string, Product> database;
public:
    InMemoryProductRepository() {
        // داده‌های اولیه (Seed Data)
        // برای تست و اجرای مثال
        database["prod_1"] = {
            "prod_1",
            "Laptop",
            1500.0,
            10
        };
        database["prod_2"] = {
            "prod_2",
            "Wireless Mouse",
            25.0,
            50
        };
    }
    std::optional<Product> findById(const std::string& id) const override {
        auto it = database.find(id);
        if (it != database.end()) {
            return it->second;
        }
        return std::nullopt;
    }

    void save(const Product& product) override {
        if (database.find(product.id) != database.end()) {
            throw std::runtime_error(
                "Product already exists. Use update instead."
            );
        }
        database[product.id] = product;
    }

    void update(const Product& product) override {
        if (database.find(product.id) == database.end()) {
            throw std::runtime_error(
                "Product not found. Cannot update."
            );
        }
        database[product.id] = product;
    }
};

} // namespace Adapters::Secondary

```

```

//src/adapters/secondary/InMemoryOrderRepository.hpp
#pragma once
#include <optional>
#include <stdexcept>
#include <string>
#include <unordered_map>
#include "src/domain/entities/Order.hpp"
#include "src/ports/outbound/OutboundPorts.hpp"

namespace Adapters::Secondary {
class InMemoryOrderRepository
    : public Ports::Outbound::IOrderRepository {
private:
    std::unordered_map<std::string, Order> database;
public:
    std::optional<Order> findById(const std::string& id) const override {
        auto it = database.find(id);
        if (it != database.end()) {
            return it->second;
        }
        return std::nullopt;
    }
    void save(const Order& order) override {
        if (database.find(order.id) != database.end()) {
            throw std::runtime_error(
                "Order already exists."
            );
        }
        database[order.id] = order;
    }
    void update(const Order& order) override {
        if (database.find(order.id) == database.end()) {
            throw std::runtime_error(
                "Order not found. Cannot update."
            );
        }
        database[order.id] = order;
    }
};

} // namespace Adapters::Secondary
// src/adapters/secondary/SimpleIdGenerator.hpp
#pragma once
#include <atomic>
#include <string>
#include "src/ports/outbound/OutboundPorts.hpp"
namespace Adapters::Secondary {
class SimpleIdGenerator: public Ports::Outbound::IIIdGenerator {
private:
    std::atomic<int> counter{1000};
public:
    std::string generate() override {
        return "ORD-" + std::to_string(counter++);
        // uuid is better option
    }
};
} // namespace Adapters::Secondary

```

```
//src/adapters/secondary/MockPaymentGateway.hpp
#pragma once
#include <cstdlib>
#include <string>
#include "src/ports/outbound/OutboundPorts.hpp"
namespace Adapters::Secondary {

class MockPaymentGateway : public Ports::Outbound::IPaymentGateway {
public:
    PaymentResult charge(const std::string& customerId, double amount) override {
        // شبیه سازی پرداخت موفق برای مبالغ زیر ۵۰۰۰ دلار
        if (amount > 5000.0) {
            return { false, "", "Insufficient funds or limit exceeded" };
        }
        return { true, "TXN-" + std::to_string(std::rand()), "" };
    }
};
} // namespace Adapters::Secondary

// src/adapters/secondary/ConsoleEmailSender.hpp
#pragma once
#include <iostream>
#include <string>
#include "src/ports/outbound/OutboundPorts.hpp"
namespace Adapters::Secondary {
class ConsoleEmailSender : public Ports::Outbound::IEmailSender {
public:
    void sendOrderConfirmation(const std::string& to, const Order& order) override {
        std::cout
            << "[EMAIL SIMULATION] "
            << "Sending confirmation to: "
            << to
            << " for Order ID: "
            << order.id
            << " | Total: $"
            << order.totalAmount()
            << "\n";
    }
};
} // namespace Adapters::Secondary
```

💡 نکته: REST و CLI هر دو از یک Use Case مشترک استفاده می کنند، منطق ایجاد سفارش هرگز تکرار نمی شود. به بیان بهتر شما می توانید UI ورودی و سرویس های خروجی مانند دیتابیس و ارسالگر ایمیل و اس ام اس و ... را تغییر دهید ولی منطق کد همیشه ثابت است!

🌱 پیاده سازی Composite Root. جایی مثل تابع main که همه آیتم ها ساخته می شوند:

```
//src/config/AppContainer.hpp
#pragma once
#include <memory>
// شامل کردن پورته ها و Use Case ها
#include "src/ports/inbound/ICreateOrderUseCase.hpp"
#include "src/application/use-cases/CreateOrderUseCase.hpp"
// شامل کردن آداپترهای ثانویه
// (پیاده سازی های واقعی)
#include "src/adapters/secondary/InMemoryProductRepository.hpp"
#include "src/adapters/secondary/InMemoryOrderRepository.hpp"
#include "src/adapters/secondary/MockPaymentGateway.hpp"
#include "src/adapters/secondary/ConsoleEmailSender.hpp"
#include "src/adapters/secondary/SimpleIdGenerator.hpp"
```

```

namespace Config {

class AppContainer {

private:
    // نگهداری نمونه‌های آداپترهای ثانویه
    // برای مدیریت چرخه حیات و اشتراک‌گذاری
    std::shared_ptr<Adapters::Secondary::InMemoryProductRepository> productRepo;

    std::shared_ptr<Adapters::Secondary::InMemoryOrderRepository> orderRepo;

    std::shared_ptr<Adapters::Secondary::MockPaymentGateway> paymentGateway;

    std::shared_ptr<Adapters::Secondary::ConsoleEmailSender> emailSender;

    std::shared_ptr<Adapters::Secondary::SimpleIdGenerator> idGenerator;
    // نگهداری نمونه‌های Use Case ها
    std::shared_ptr<Application::UseCases::CreateOrderUseCase> createOrderUseCase;
public:
    AppContainer() {
        // ساخت آداپترهای ثانویه (زیرساخت)
        productRepo =
            std::make_shared<Adapters::Secondary::InMemoryProductRepository>();

        orderRepo = std::make_shared<Adapters::Secondary::InMemoryOrderRepository>();

        paymentGateway = std::make_shared<Adapters::Secondary::MockPaymentGateway>();

        emailSender = std::make_shared<Adapters::Secondary::ConsoleEmailSender>();

        idGenerator = std::make_shared<Adapters::Secondary::SimpleIdGenerator>();
        // تزریق وابستگی‌ها به Use Case ها
        // نکته مهم:
        // از عملگر * برای تبدیل shared_ptr به Reference
        // استفاده می‌کنیم، زیرا Constructor مربوط به Use Case
        // بر اساس (&) Reference طراحی شده است.
        createOrderUseCase =
            std::make_shared<Application::UseCases::CreateOrderUseCase>(
                *productRepo,
                *orderRepo,
                *emailSender,
                *paymentGateway,
                *idGenerator);
    }
    // ارائه دسترسی به Use Case ها برای لایه
    // Presentation (Primary Adapters)
    Application::UseCases::CreateOrderUseCase& getCreateOrderUseCase() {
        return *createOrderUseCase;
    }
};

} // namespace Config

```



```

#include <iostream>
#include <vector>
#include "src/config/AppContainer.hpp"
#include "src/domain/exceptions/product_exceptions.hpp"

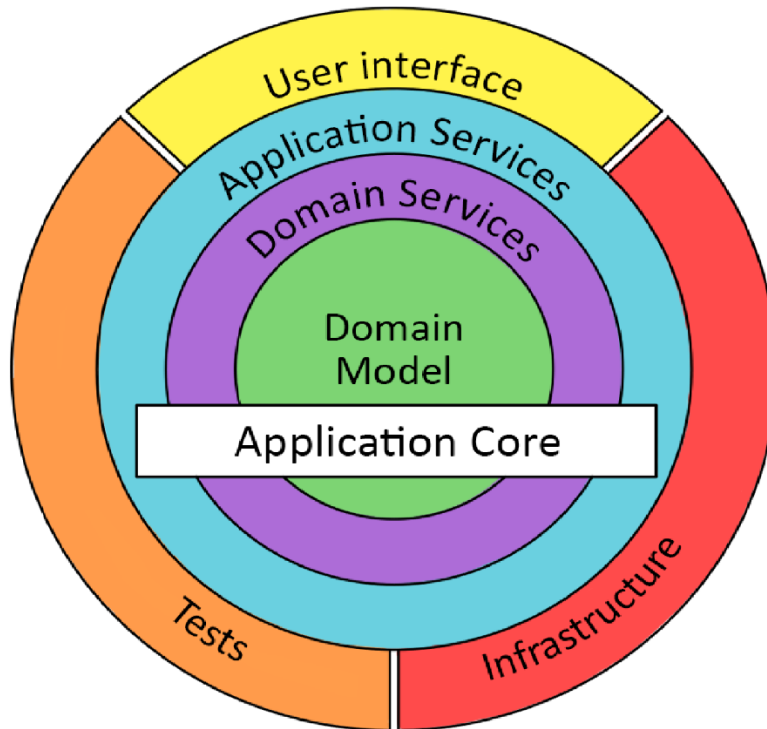
int main() {
    try {
        // راه اندازی کانتینر
        // (تمام تزریق ها اینجا انجام می شود)
        Config::AppContainer container;
        // دریافت Use Case آماده به کار
        auto& useCase = container.getCreateOrderUseCase();
        // ساخت درخواست از سمت کلاینت
        // (شبه سازی شده Primary Adapter)
        Ports::Inbound::CreateOrderRequest request;
        request.customerId = "CUST-99";
        request.items = {
            {"prod_1", 2}, // لیپ تاپ (موجودی کافی است)
            {"prod_2", 1}  // ماوس (موجودی کافی است)
        };
        std::cout << "--- Starting Order Creation Process ---\n";
        // اجرای منطق کسب و کار
        Ports::Inbound::CreateOrderResponse response = useCase.execute(request);
        // نمایش نتیجه به کلاینت
        if (response.success) {
            std::cout << "\nSUCCESS!\n";
            std::cout
                << "Order ID: "
                << response.orderId << "\n";
            std::cout
                << "Total Amount: $"
                << response.totalAmount << "\n";
            std::cout
                << "Message: "
                << response.message << "\n";
        } else {
            std::cout << "\nFAILED!\n";
            std::cout
                << "Message: "
                << response.message << "\n";
        }
    }
    // مدیریت خطاهای دامنه
    catch (const DomainException& e) {
        std::cerr
            << "\nDOMAIN ERROR: "
            << e.what() << "\n";
        return 1;
    }
    // مدیریت سایر خطاهای غیرمنتظره
    catch (const std::exception& e) {
        std::cerr
            << "\nSYSTEM ERROR: "
            << e.what() << "\n";
        return 1;
    }
    return 0;
}

```

از مزایای مهم معماری Hexagonal همانند معماری های مشابهش اینست که در این معماری، از دید هسته، فرقی نمی کند درخواست از یک رابط وب می آید یا از یک تست خودکار! چون هر دو صرفاً یک Driving Adapter هستند و تمرکز اصلی روی «قابلیت جایگزینی و تست پذیری از طریق درگاه های متقارن» است.

این معماری در سال ۲۰۰۸ توسط جفری پالرمو (Jeffrey Palermo) معرفی شد. او مستقیماً از دنیای DDD الهام گرفت و تأکید ویژه‌ای روی جایگاه مرکزی مدل دامنه داشت.

 **ساختار حلقوی:** Onion با ساختار لایه ای تو در تو (شبیه به پیاز) با چند حلقه هم‌مرکز نمایش داده می‌شود که از بیرون به داخل این گونه‌اند:



```
Infrastructure (UI, DB, External Services)
└─> Application Services
    └─> Domain Services
        └─> Domain Model (Entities, Value Objects)
```

معماری Onion از لایه های زیر تشکیل می شود:

- 1- **Domain Model:** قلب این معماری است که کاملاً مستقل است و مدلها و قوانین کسب و کار و به عبارتی Entity ها و ValueObject ها در دل این قسمت قرار دارد! این لایه مسئول قوانین و منطق کسب و کار خالص است و نباید به فناوری یا زیرساخت خاصی وابسته باشد.
- 2- **Domain Services:** منطق کسب و کار و دامنه که به چند Entity نیاز دارد. مثال PricingService: قیمت گذاری چند محصول
- 3- **Application Services:** لایه «لاغر» هماهنگ کننده که خودش منطق تولید نمی کند! به عبارت دیگری این لایه، لایه ای لاغر و هماهنگ کننده است و خودش منطق اصلی کسب و کار را تولید نمی کند؛ بلکه داده را دریافت می کند، Domain را فراخوانی می کند و نتیجه را برمی گرداند!

4- Infrastructure/User Interface(UI)/Tests: لایه بیرونی که وابسته به لایه های داخلی است و شامل UI و تست ها و دیتابیس و

سرویس های ارتباط با سیستم های خارج از نرم افزار است! این لایه مسئول ارتباط با فناوری ها و سرویس های واقعی است؛ مانند دیتابیس های SQL/NoSQL، سرویس های خارجی همانند سیستم های پیام رسان و سایر جزئیات زیرساختی .

 **قانون اصلی Onion:** هر حلقه (لایه) فقط می تواند به حلقه های داخلی تر از خودش وابسته باشد و هرگز نباید به لایه های بیرونی خود وابسته باشد. قانون معروف Onion همانند Hexagonal این است که صراحتاً می گوید Interface های دسترسی به داده (Repository) باید در لایه داخلی تعریف شوند و پیاده سازی واقعی شان در بیرونی ترین لایه (Infrastructure) قرار گیرد. Onion روی جایگاه مرکزی مدل دامنه در سبک DDD تمرکز دارد.

در مرکز این معماری، Domain Model قرار دارد. هسته مرکزی باید کاملاً مستقل باشد و هیچ وابستگی ای به لایه های بیرونی نداشته باشد؛ نه به دیتابیس، نه به فریم ورک و نه حتی به لایه Application! نام گذاری لایه های Onion کاملاً متأثر از DDD است و تا حدود زیادی سعی کرده پایبند به اصول و نام گذاری های DDD بماند!

جایگاه Repository و اصل DIP

یکی از مهم ترین نکات Onion این است که Interface های مربوط به دسترسی به داده (Repository) در لایه های داخلی و نزدیک به Domain تعریف می شوند، در حالی که پیاده سازی واقعی آن ها در لایه بیرونی تر، یعنی Infrastructure، قرار می گیرد.

به این ترتیب، وابستگی ها از بیرون به داخل شکل می گیرند و این ساختار در واقع پیاده سازی مشخصی از اصل Dependency Inversion Principle (DIP) در پروژه های مبتنی بر DDD است.

 **پیاده سازی مثال فروشگاه قبل این بار با معماری Onion:** در ابتدا ساختار فایل های این پروژه را در شکل بعدی مشاهده می فرمایید و سپس کد نوشته شده فروشگاه را به سبک Onion Architecture قرار می دهیم!

```
//-----  
//Domain Model Layer  
//-----  
//src/domain/exceptions/product_exceptions.hpp  
#pragma once  
#include <exception>  
#include <string>  
#include <utility>  
//  
// Domain Exceptions  
//  
struct DomainException : std::exception {  
    std::string msg;  
  
    explicit DomainException(std::string message)  
        : msg(std::move(message)) {}  
  
    const char* what() const noexcept override {  
        return msg.c_str();  
    }  
};
```

```

struct InsufficientStockException : DomainException {
    InsufficientStockException(
        const std::string& productId,int requested,int available):
        DomainException(
            "Product " + productId +
            ": requested " + std::to_string(requested) +
            " but only " + std::to_string(available) +
            " available"
        ) {}
};

struct ProductNotFoundException : DomainException {
    explicit ProductNotFoundException(const std::string& id)
        : DomainException( "Product not found: " + id) {}
};
//src/domain/entities/Order.hpp
#pragma once

#include <string>
#include <vector>

#include "src/domain/exceptions/product_exceptions.hpp"
// -----
// Product
// -----
struct Product {
    std::string id;
    std::string name;
    double price;
    int stock; // تعداد موجودی کالا در انبار
};
// -----
// Order Item
// -----
struct OrderItem {
    std::string productId;
    int quantity; // تعداد کالا در سفارش مشتری
    double unitPrice;
    /*
    قیمت واحد، قیمت لحظه خرید توسط کاربر است که شاید با قیمت
    کنونی فرق داشته باشد.
    unitPrice ، قیمت به روز فعلی محصول است؛ اما Product قیمت
    لحظه خرید توسط خریدار است.
    فرض کنید پس از خرید توسط مشتری، قیمت کنونی محصول گران تر
    به جهت مشکلات دیتابیس یا Order یا ارزان تر شود و هنوز
    هر نوع تأخیر ثبت نشده باشد
    ثبت شود Order در این حالت باید قیمت زمان خرید در
    */
    double total() const {
        return unitPrice * quantity;
    }
};
// -----
// Order
// -----
struct Order {
    std::string id;
    std::string customerId;
    std::vector<OrderItem> items;
};

```

```

enum class Status {
    Pending,
    Confirmed,
    Shipped,
    Cancelled
};

Status status = Status::Pending;

double totalAmount() const {
    double total = 0;
    for (const auto& item : items) {
        total += item.total();
    }
    return total;
}
// یک متد ساده برای نشان دادن منطق دامنه
void addItem(const OrderItem& item, int currentStock) {
    if (status != Status::Pending) {
        throw DomainException("Cannot add items to a non-pending order");
    }
    if (item.quantity > currentStock) {
        throw
            InsufficientStockException(item.productId, item.quantity, currentStock);
    }
    items.push_back(item);
}
};

//-----
//Domain Services Layer
//-----
//src/domain/services/PricingService.hpp
#pragma once
#include <vector>
#include "src/domain/entities/Order.hpp"

class PricingService {
public:
    static double calculateTotal(const std::vector<OrderItem>& items);
    static double applyDiscount(double amount, double discountPercent);
    static double calculateTax(double amount, double taxRate);
};

//-----
//Application Services Layer
//-----
//src/application/ports/inbound/ICreateOrderUseCase.hpp
#pragma once

#include <string>
#include <vector>

#include "src/domain/entities/Order.hpp"

namespace Ports::Inbound {

struct CreateOrderRequest {
    std::string customerId;
    std::vector<OrderItem> items;
};
};

```

```

struct CreateOrderResponse {
    bool success;
    std::string orderId;
    std::string message;
    double totalAmount;
};

class ICreateOrderUseCase {
public:
    virtual ~ICreateOrderUseCase() = default;

    virtual CreateOrderResponse execute(const CreateOrderRequest& request) = 0;
};

} // namespace Ports::Inbound

//src/application/ports/inbound/IGetOrderUseCase.hpp
#pragma once

#include <optional>
#include <string>

#include "src/domain/entities/Order.hpp"

namespace Ports::Inbound {

class IGetOrderUseCase {
public:
    virtual ~IGetOrderUseCase() = default;
    // بازگرداندن optional نشان می‌دهد
    // که ممکن است سفارش پیدا نشود
    virtual std::optional<Order> execute(const std::string& orderId) = 0;
};

} // namespace Ports::Inbound
//src/application/ports/inbound/ICancelOrderUseCase.hpp
#pragma once
#include <string>

namespace Ports::Inbound {

class ICancelOrderUseCase {
public:
    virtual ~ICancelOrderUseCase() = default;
    // در صورت موفقیت true و در صورت خطا
    // (مثلاً سفارش قبلاً کنسل شده)
    // false یا Exception برمی‌گرداند
    virtual bool execute(const std::string& orderId) = 0;
};

} // namespace Ports::Inbound
//src/application/ports/outbound/OutboundPorts.hpp
#pragma once
#include <optional>
#include <string>
#include "src/domain/entities/Order.hpp"

```

```

namespace Ports::Outbound {
// _____
// Product Repository
// _____
class IProductRepository {
public:
    virtual ~IProductRepository() = default;

    virtual std::optional<Product> findById(const std::string& id) const = 0;

    virtual void save(const Product& product) = 0;

    virtual void update(const Product& product) = 0;
};
// _____
// Order Repository
// _____
class IOrderRepository {
public:
    virtual ~IOrderRepository() = default;

    virtual std::optional<Order> findById(const std::string& id) const = 0;

    virtual void save(const Order& order) = 0;

    virtual void update(const Order& order) = 0;
};
// _____
// Email Sender
// _____
class IEmailSender {
public:
    virtual ~IEmailSender() = default;

    virtual void sendOrderConfirmation(const std::string& to, const Order& order) = 0;
};
// _____
// Payment Gateway
// _____
class IPaymentGateway {
public:
    struct PaymentResult {
        bool success;
        std::string transactionId;
        std::string errorMessage;
    };

    virtual ~IPaymentGateway() = default;

    virtual PaymentResult charge(const std::string& customerId, double amount) = 0;
};
// _____
// ID Generator
// _____
class IIdGenerator {
public:
    virtual ~IIdGenerator() = default;

    virtual std::string generate() = 0;
};
} // namespace Ports::Outbound

```



```

//src/application/use-cases/CreateOrderUseCase.hpp
#pragma once

#include "src/application/ports/inbound/ICreateOrderUseCase.hpp"
#include "src/application/ports/outbound/OutboundPorts.hpp"

namespace Application::UseCases {

class CreateOrderUseCase: public Ports::Inbound::ICreateOrderUseCase {

private:
    // استفاده از (& Reference عالی است،
    // زیرا تضمین می‌کند وابستگی‌ها null نیستند)
    Ports::Outbound::IProductRepository& productRepo_;
    Ports::Outbound::IOrderRepository& orderRepo_;
    Ports::Outbound::IEmailSender& emailSender_;
    Ports::Outbound::IPaymentGateway& payment_;
    Ports::Outbound::IIIdGenerator& idGen_;

public:
    CreateOrderUseCase(
        Ports::Outbound::IProductRepository& p,
        Ports::Outbound::IOrderRepository& o,
        Ports::Outbound::IEmailSender& e,
        Ports::Outbound::IPaymentGateway& pay,
        Ports::Outbound::IIIdGenerator& id
    )
        : productRepo_(p),
          orderRepo_(o),
          emailSender_(e),
          payment_(pay),
          idGen_(id) {}

    Ports::Inbound::CreateOrderResponse execute(
        const Ports::Inbound::CreateOrderRequest& request) override;
};

} // namespace Application::UseCases

//src/application/use-cases/CreateOrderUseCase.cpp
#include "src/application/use-cases/CreateOrderUseCase.hpp"

#include "src/domain/entities/Order.hpp"
#include "src/domain/exceptions/product_exceptions.hpp"

namespace Application::UseCases {

Ports::Inbound::CreateOrderResponse CreateOrderUseCase::execute(
    const Ports::Inbound::CreateOrderRequest& request) {

    // ساخت شیء Order در حافظه
    // (هنوز در دیتابیس ذخیره نشده)
    Order order;

    order.id = idGen_.generate();
    order.customerId = request.customerId;
    order.status = Order::Status::Pending;
    // بررسی موجودی و ساخت آیتم‌ها
    // (بدون تغییر در دیتابیس)
    for (const auto& reqItem : request.items) {
        auto productOpt = productRepo_.findById(reqItem.productId);
    }
}

```

```

        if (!productOpt) {
            throw ProductNotFoundException(reqItem.productId);
        }
        Product& product = *productOpt;
        // استفاده از رفرنس برای قابلیت تغییر
        if (product.stock < reqItem.quantity) {
            throw InsufficientStockException(product.id, reqItem.quantity, product.stock);
        }

        order.items.push_back({reqItem.productId, reqItem.quantity, product.price});
    }
    // انجام پرداخت
    // (قبل از هرگونه تغییر در دیتابیس)
    auto payResult = payment_.charge(request.customerId, order.totalAmount());
    if (!payResult.success) {
        // پرداخت ناموفق بود،
        // هیچ تغییری در دیتابیس ایجاد نشده است
        return {false, "", "Payment failed: " + payResult.errorMessage, 0.0};
    }
    // پرداخت موفق بود:
    // اکنون تغییرات در دیتابیس اعمال می‌شوند
    order.status = Order::Status::Confirmed;

    orderRepo_.save(order);
    // به روزرسانی موجودی محصولات می‌تواند
    // در یک Transaction دیتابیس باشد
    for (const auto& orderItem : order.items) {
        auto productOpt = productRepo_.findById(orderItem.productId);
        if (productOpt) {
            productOpt->stock -= orderItem.quantity;
            productRepo_.update(*productOpt);
        }
    }
    // ارسال ایمیل تأیید
    // در واقعیت ایمیل باید از روی User گرفته شود.
    // اینجا برای سادگی فرض شده customerId همان ایمیل است.
    emailSender_.sendOrderConfirmation(request.customerId, order);
    // بازگرداندن پاسخ موفق
    return {true, order.id, "Order created successfully", order.totalAmount()};
}

} // namespace Application::UseCases
//-----
//Infrastructure-UI-Tests Layer
//-----
//src/infrastructure/adapters/secondary/InMemoryProductRepository.hpp
#pragma once

#include <optional>
#include <stdexcept>
#include <string>
#include <unordered_map>

#include "src/domain/entities/Order.hpp"
#include "src/application/ports/outbound/OutboundPorts.hpp"

namespace Infrastructure::Adapters::Secondary {

class InMemoryProductRepository : public Ports::Outbound::IProductRepository {

```

```

private:
    //Map شبیه سازی دیتابیس با استفاده از یک
    std::unordered_map<std::string, Product> database;
public:
    InMemoryProductRepository() {
        // داده های اولیه (Seed Data)
        // برای تست و اجرای مثال
        database["prod_1"] = {"prod_1", "Laptop", 1500.0, 10};

        database["prod_2"] = {"prod_2", "Wireless Mouse", 25.0, 50};
    }

    std::optional<Product> findById(const std::string& id) const override {
        auto it = database.find(id);
        if (it != database.end()) {
            return it->second;
        }
        return std::nullopt;
    }

    void save(const Product& product) override {
        if (database.find(product.id) != database.end()) {
            throw std::runtime_error("Product already exists. Use update instead.");
        }

        database[product.id] = product;
    }

    void update(const Product& product) override {
        if (database.find(product.id) == database.end()) {
            throw std::runtime_error(
                "Product not found. Cannot update."
            );
        }
        database[product.id] = product;
    }
};

} // namespace Infrastructure::Adapters::Secondary
//src/infrastructure/adapters/secondary/InMemoryOrderRepository.hpp
#pragma once

#include <optional>
#include <stdexcept>
#include <string>
#include <unordered_map>

#include "src/domain/entities/Order.hpp"
#include "src/application/ports/outbound/OutboundPorts.hpp"

namespace Infrastructure::Adapters::Secondary {

class InMemoryOrderRepository: public Ports::Outbound::IOrderRepository {
private:
    std::unordered_map<std::string, Order> database;

```

```

public:
    std::optional<Order> findById(const std::string& id) const override {
        auto it = database.find(id);
        if (it != database.end()) {
            return it->second;
        }
        return std::nullopt;
    }

    void save(const Order& order) override {

        if (database.find(order.id) != database.end()) {
            throw std::runtime_error("Order already exists.");
        }
        database[order.id] = order;
    }

    void update(const Order& order) override {
        if (database.find(order.id) == database.end()) {

            throw std::runtime_error("Order not found. Cannot update.");
        }
        database[order.id] = order;
    }
};

} // namespace Infrastructure::Adapters::Secondary

//src/infrastructure/adapters/secondary/SimpleIdGenerator.hpp
#pragma once

#include <atomic>
#include <string>

#include "src/application/ports/outbound/OutboundPorts.hpp"

namespace Infrastructure::Adapters::Secondary {

class SimpleIdGenerator: public Ports::Outbound::IIdGenerator {
private:
    std::atomic<int> counter{1000};

public:
    std::string generate() override {
        return "ORD-" +std::to_string(counter++); // uuid is better option
    }
};

} // namespace Infrastructure::Adapters::Secondary

//src/infrastructure/adapters/secondary/MockPaymentGateway.hpp
#pragma once

#include <cstdlib>
#include <string>
#include "src/application/ports/outbound/OutboundPorts.hpp"
namespace Infrastructure::Adapters::Secondary {

class MockPaymentGateway: public Ports::Outbound::IPaymentGateway {

```

```

public:
    PaymentResult charge(const std::string& customerId,double amount) override {
        // شبیه‌سازی پرداخت موفق
        // برای مبالغ زیر ۵۰۰۰ دلار
        if (amount > 5000.0) {
            return {false,"","Insufficient funds or limit exceeded"};
        }

        return {true,"TXN-" +std::to_string(std::rand()),""};
    }
};

} // namespace Infrastructure::Adapters::Secondary

//src/infrastructure/adapters/secondary/ConsoleEmailSender.hpp
#pragma once

#include <iostream>
#include <string>
#include "src/application/ports/outbound/OutboundPorts.hpp"

namespace Infrastructure::Adapters::Secondary {

class ConsoleEmailSender: public Ports::Outbound::IEmailSender {
public:
    void sendOrderConfirmation(const std::string& to,const Order& order) override {

        std::cout<< "[EMAIL SIMULATION] "
            << "Sending confirmation to: "
            << to
            << " for Order ID: "
            << order.id
            << " | Total: $"
            << order.totalAmount()<< "\n";
    }
};

} // namespace Infrastructure::Adapters::Secondary
//-----
// Composit Root
//-----
//src/config/AppContainer.hpp
#pragma once

#include <memory>

#include "src/application/use-cases/CreateOrderUseCase.hpp"
#include "src/infrastructure/adapters/secondary/InMemoryProductRepository.hpp"
#include "src/infrastructure/adapters/secondary/InMemoryOrderRepository.hpp"
#include "src/infrastructure/adapters/secondary/MockPaymentGateway.hpp"
#include "src/infrastructure/adapters/secondary/ConsoleEmailSender.hpp"
#include "src/infrastructure/adapters/secondary/SimpleIdGenerator.hpp"

namespace Config {
class AppContainer {
private:
    //نگهداری نمونه‌های Infrastructure
    // برای مدیریت چرخه حیات و اشتراک‌گذاری
    std::shared_ptr<Infrastructure::Adapters::Secondary::InMemoryProductRepository> productRepo;

    std::shared_ptr<Infrastructure::Adapters::Secondary::InMemoryOrderRepository> orderRepo;

```

```

std::shared_ptr<Infrastructure::Adapters::Secondary::MockPaymentGateway> paymentGateway;

std::shared_ptr<Infrastructure::Adapters::Secondary::ConsoleEmailSender> emailSender;

std::shared_ptr<Infrastructure::Adapters::Secondary::SimpleIdGenerator> idGenerator;
//Use Case نگهداری نمونه
std::shared_ptr<Application::UseCases::CreateOrderUseCase> createOrderUseCase;

public:
AppContainer() {
    //Infrastructure آداپترهای ساخت
    productRepo
        =std::make_shared<Infrastructure::Adapters::Secondary::InMemoryProductRepository>();

    orderRepo
        =std::make_shared<Infrastructure::Adapters::Secondary::InMemoryOrderRepository>();

    paymentGateway
        =std::make_shared<Infrastructure::Adapters::Secondary::MockPaymentGateway>();

    emailSender
        =std::make_shared<Infrastructure::Adapters::Secondary::ConsoleEmailSender>();

    idGenerator =std::make_shared<Infrastructure::Adapters::Secondary::SimpleIdGenerator>();
    //Use Case تزریق وابستگی‌ها به
    //Use Case به Interface ها وابسته است،
    //ولی اینجا Implementation های واقعی
    //به آنها تزریق می‌شوند.
    createOrderUseCase =std::make_shared<Application::UseCases::CreateOrderUseCase>(
        *productRepo,
        *orderRepo,
        *emailSender,
        *paymentGateway,
        *idGenerator
    );
}
//Use Case ارائه دسترسی به
//Presentation برای لایه
Application::UseCases::CreateOrderUseCase& getCreateOrderUseCase() {
    return *createOrderUseCase;
}
};

} // namespace Config
//Presentation
//src/presentation/ConsoleApp.hpp
#pragma once

#include <iostream>
#include "src/application/ports/inbound/ICreateOrderUseCase.hpp"

namespace Presentation {

class ConsoleApp {
private:
    Ports::Inbound::ICreateOrderUseCase& createOrderUseCase_;

public:
    explicit ConsoleApp(Ports::Inbound::ICreateOrderUseCase& useCase)
        : createOrderUseCase_(useCase) {}
}

```

```

void run() {
    // ساخت درخواست از سمت کلاینت
    // (Primary Adapter شبیه سازی شده)
    Ports::Inbound::CreateOrderRequest request;
    request.customerId = "CUST-99";
    request.items = {
        {"prod_1", 2},
        {"prod_2", 1}
    };
    std::cout<< "--- Starting Order Creation Process ---\n";
    // اجرای Use Case
    auto response = createOrderUseCase_.execute(request);
    // نمایش نتیجه به کلاینت
    if (response.success) {
        std::cout<< "\nSUCCESS!\n";

        std::cout<< "Order ID: "<< response.orderId<< "\n";

        std::cout<< "Total Amount: $"<< response.totalAmount<< "\n";

        std::cout<< "Message: "<< response.message<< "\n";
    } else {
        std::cout<< "\nFAILED!\n";

        std::cout<< "Message: "<< response.message<< "\n";
    }
}

};

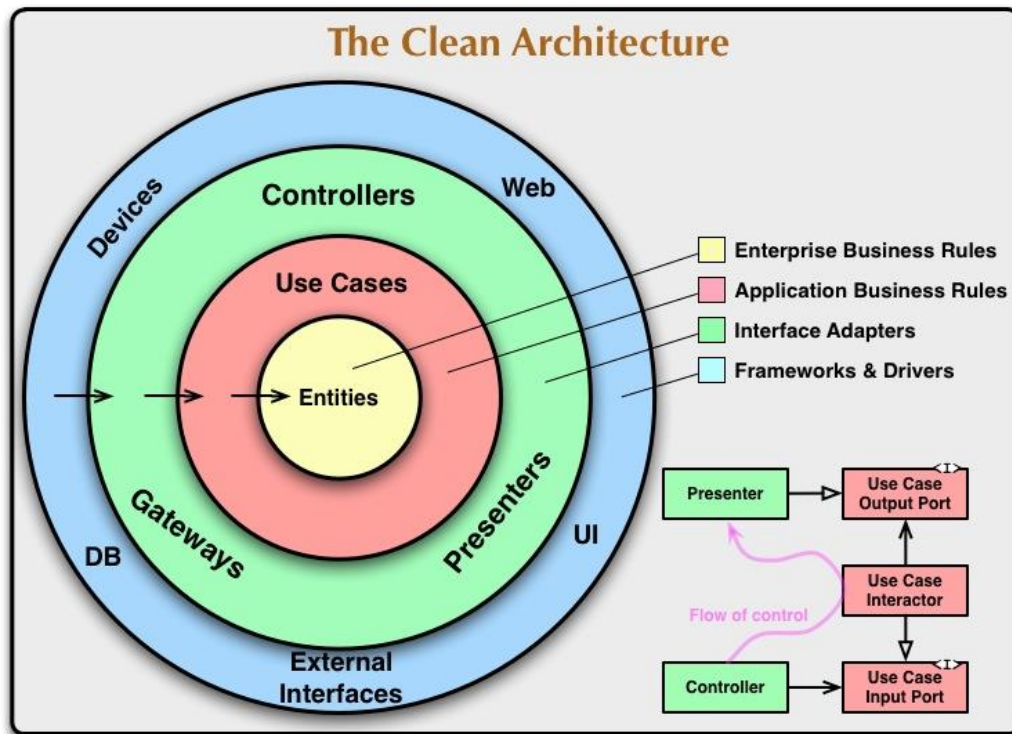
} // namespace Presentation
//main.cpp
#include <iostream>
#include "src/config/AppContainer.hpp"
#include "src/presentation/ConsoleApp.hpp"
#include "src/domain/exceptions/product_exceptions.hpp"

int main() {
    try {
        // راه اندازی Composition Root
        // تمام Dependency Injection ها
        // در AppContainer انجام می شوند
        Config::AppContainer container;
        // Presentation به Inbound Port وابسته است.
        // Use Case واقعی از طریق Container
        // به آن تزریق می شود
        Presentation::ConsoleApp app(container.getCreateOrderUseCase());
        app.run();
    }
    // مدیریت خطاهای دامنه
    catch (const DomainException& e) {
        std::cerr<< "\nDOMAIN ERROR: "<< e.what()<< "\n";
        return 1;
    }
    // مدیریت سایر خطاهای غیرمنتظره
    catch (const std::exception& e) {
        std::cerr<< "\nSYSTEM ERROR: "<< e.what()<< "\n";
        return 1;
    }

    return 0;
}

```

این معماری در سال ۲۰۱۲ توسط رابرت مارتین (Uncle Bob) معرفی شد. مارتین به صراحت می گوید Clean Architecture ترکیبی آگاهانه از ایده های پیشین (Onion و Hexagonal) است؛ هدف او تدوین یک زبان مشترک برای این خانواده از الگوها بود. در اصل، Onion Architecture پیشرو Clean Architecture محسوب می شود. Clean تقریباً یک نسخه ی تکامل یافته و رسمی تر از همان ایده است.



```
Frameworks & Drivers (Web, DB, UI, External Devices)
└> Interface Adapters (Controllers, Presenters, Gateways)
    └> Use Cases (Application Business Rules)
        └> Entities (Enterprise Business Rules)
```

🎯 هدف نهایی این معماری، سیستمی است که:

- مستقل از فریم ورک باشد!
- بدون UI، دیتابیس یا وب سرور، قابل تست باشد!
- مستقل از رابط کاربری و دیتابیس باشد!
- مستقل از هر عامل یا سیستم خارجی باشد!

🏠 لایه های معماری Clean:

- 1- **Entities**: قوانین کسب و کار سطح بالا و مستقل از Application و پایدارترین بخش سیستم است. حتی اگر برنامه عوض شود، این لایه همچنان صادق است. این لایه معادل تقریبی Domain Model در Domain و Onion در DDD است.
- 2- **Use Cases**: منطق مخصوص برنامه که Entity ها را برای برآوردن سناریوهای خاص هماهنگ می کند. معادل Application/Domain Services در Onion یا Driving Port در Hexagonal است و مفهوم کاربردی این لایه یعنی «این برنامه

دقیقاً چه کاری برای کاربر انجام می‌دهد و برای این کار چه متدی با چه ورودی‌هایی باید اجرا شود و چه خروجی را به عنوان نتیجه پردازش می‌دهد».

- 3- **Interface Adapters**: مبدل بین فرمت Use Case و فرمت دنیای سمت مشتریان و کلاینت هاست. شامل Controller ها، Presenter ها و Gateway ها! در اصل این لایه مبدل‌هایی دارد که داده را بین فرمت مناسب Use Case/Entity و فرمت مناسب دنیای بیرون (سمت Client ها) تبدیل می‌کنند (که در اصل همان Repository Interface ها هستند اما پیاده‌سازی آن‌ها بیرون است)!
- 4- **Frameworks & Drivers**: بیرونی‌ترین لایه است و جزئیات فنی خام در این بخش پیاده‌سازی می‌شوند. مانند وب فریم‌ورک، ORM و درایورهای دیتابیس و UI و تست!

Clean روی قانون وابستگی به عنوان یک اصل عمومی و مستقل از DDD تمرکز دارد و چتری برای Onion و Hexagonal است!

🏠 قانون وابستگی (The Dependency Rule)

وابستگی‌های کد سورس فقط می‌توانند به سمت داخل اشاره کنند. هیچ چیز در یک حلقه داخلی نباید چیزی درباره یک حلقه بیرونی‌تر بداند.

این دقیقاً همان قانونی است که در Onion و Hexagonal و به طور کلی در DDD هم دیدیم! فقط با اسامی لایه‌های متفاوت. با اینکه تناظر بالا نشان می‌دهد پوسته این سه معماری تقریباً یکسان است، چند تفاوت واقعی و مهم هم وجود دارد! مثلاً در تعداد و صراحت لایه‌ها! Clean Architecture دقیقاً چهار حلقه را نام می‌برد و مارتین صراحتاً می‌گوید تعداد حلقه‌ها می‌تواند بیشتر یا کمتر از ۴ باشد؛ عدد ۴ صرفاً نمونه است. Onion هم معمولاً با ۳ تا ۴ حلقه ترسیم می‌شود اما پالرمو تعداد را ثابت نمی‌داند. Hexagonal اصلاً درباره «تعداد لایه» صحبت نمی‌کند و فقط یک مرز دوگانه ساده دارد: (هسته در برابر همه چیزهای دیگر، به علاوه مفهوم Port/Adapter).

مثال کامل معماری Clean در ++C: در معماری Clean، هدف اصلی جداسازی قوانین کسب‌وکار از جزئیات تکنولوژیک است. در این مثال یک سیستم فروشگاهی را پیاده‌سازی می‌کنیم که درخواست HTTP از کاربر دریافت می‌شود، توسط Handler و Controller به Use Case می‌رسد، Use Case قوانین کاربردی را اجرا می‌کند و برای دسترسی به داده، پرداخت و ارسال ایمیل از طریق Outbound Port ها با دنیای بیرون ارتباط می‌گیرد.

در این ساختار، چهار لایه در نظر گرفته شده است: لایه Domain، لایه Application، لایه Interface Adapters و لایه Infrastructure! وابستگی‌ها باید به سمت مرکز حرکت کنند؛ بنابراین Domain نباید هیچ شناختی از HTTP، Database، Payment Provider یا Email Service داشته باشد.

توجه: به جهت اهمیت Clean Architecture، توضیحات این بخش را تفکیک شده تر و کامل تر قرار می‌دهیم!

- 1- **لایه Entity یا همان Domain؛ قوانین اصلی کسب‌وکار:** این لایه داخلی‌ترین لایه است و هیچ اطلاعی از Application، HTTP، Database، Payment Provider یا Email ندارد! Entity ها، Value Object ها، Aggregate ها و Exception های مربوط به قوانین دامنه در این لایه قرار می‌گیرند.

:Money.hpp

```
#pragma once
#include <stdexcept>
```

```

namespace domain {
class Money {
    long long cents_;
public:
    explicit Money(long long cents = 0) : cents_(cents) {
        if (cents < 0) throw std::invalid_argument("Money cannot be negative");
    }

    long long cents() const { return cents_; }

    Money operator+(const Money& other) const {return Money(cents_ + other.cents_); }

    Money operator*(int quantity) const { return Money(cents_ * quantity); }
};
}

```

DomainException.hpp

```

#pragma once
#include <stdexcept>

namespace domain {

class DomainException : public std::runtime_error {
public:
    using std::runtime_error::runtime_error;
};

}

```

Product.hpp

```

#pragma once
#include <string>
#include "../value_objects/Money.hpp"

namespace domain {

class Product {
    int id_;
    std::string name_;
    Money price_;
public:
    Product(int id, std::string name, Money price)
        : id_(id), name_(std::move(name)), price_(price) {}

    int id() const { return id_; }

    const std::string& name() const { return name_; }

    const Money& price() const { return price_; }
};

}

```

```
#pragma once
#include "Product.hpp"

namespace domain {

class OrderLine {
    int productId_;
    std::string productName_;
    Money unitPrice_;
    int quantity_;
public:
    OrderLine(const Product& product, int quantity)
        : productId_(product.id()), productName_(product.name()),
          unitPrice_(product.price()), quantity_(quantity) {
        if (quantity <= 0) throw DomainException("Quantity must be positive");
    }

    Money total() const { return unitPrice_ * quantity_; }

    int productId() const { return productId_; }

    int quantity() const { return quantity_; }
};

}
```

Order.hpp به عنوان Aggregate Root: در این مثال Order یک Aggregate Root است. نکته مهم این است که Aggregate Root نباید پیاده سازی کامل Repository ها را بشناسد! Repository متعلق به مرز Application (بیرونی ترین لایه) است، نه Domain!

```
#pragma once
#include <vector>
#include "OrderLine.hpp"

namespace domain {

enum class OrderStatus { Draft, Confirmed, Paid };

class Order {
    int id_;
    int customerId_;
    std::vector<OrderLine> lines_;
    OrderStatus status_ = OrderStatus::Draft;
public:
    Order(int id, int customerId) : id_(id), customerId_(customerId) {}

    void addLine(const Product& product, int quantity) {
        if (status_ != OrderStatus::Draft)
            throw DomainException("Cannot modify confirmed order");
        lines_.emplace_back(product, quantity);
    }
}
```

```

    Money total() const {
        Money result;
        for (const auto& line : lines_) result = result + line.total();
        return result;
    }

    void confirm() {
        if (lines_.empty())
            throw DomainException("Order must contain at least one product");
        status_ = OrderStatus::Confirmed;
    }

    void markAsPaid() {
        if (status_ != OrderStatus::Confirmed)
            throw DomainException("Only confirmed orders can be paid");
        status_ = OrderStatus::Paid;
    }

    int id() const { return id_; }

    int customerId() const { return customerId_; }

    OrderStatus status() const { return status_; }

    const std::vector<OrderLine>& lines() const { return lines_; }
};

```

۲. لایه **Application** به عنوان قوانین کاربردی سیستم: این لایه، Use Case های سیستم و Interface هایی را در خود دارد که Use Case برای ارتباط با بیرون نیاز دارد! Application، اقلام لایه Domain را می شناسد اما Domain نباید Application را بشناسد (نباید include کند).

- **Inbound Port ها:** به زبان ساده و عامیانه، شابلونی که به لایه 3 که کاربران (کلاینت ها) با آن ارتباط دارند می گوید، من دقیقاً چه سرویس هایی را ارائه می دهم و برای این سرویس ها، چه ورودی هایی را می گیرم و چه خروجی هایی را می دهم!

: CreateOrderUseCase.hpp

```

#pragma once
#include "../models/CreateOrderRequest.hpp"
#include "../models/CreateOrderResponse.hpp"

namespace application {

class CreateOrderUseCase {
public:
    virtual ~CreateOrderUseCase() = default;
    virtual CreateOrderResponse execute(const CreateOrderRequest&) = 0;
};

}

```

Inbound Port مشخص می‌کند که دنیای بیرون چگونه می‌تواند یک Use Case را اجرا کند! Controller یا هر Primary Adapter دیگری از طریق این Interface با Use Case ارتباط برقرار می‌کند.

:CreateOrderRequest.hpp

```
#pragma once
#include <vector>
namespace application {

struct CreateOrderRequest {
    int customerId;
    std::vector<std::pair<int, int>> products; // productId, quantity
};

}
```

این Request، HTTP Request نیست. این مدل متعلق به Application است و Controller مدل HTTP را به آن تبدیل می‌کند.

:CreateOrderResponse.hpp

```
#pragma once
namespace application {
struct CreateOrderResponse {
    int orderId;
    long long totalCents;
    bool paid;
};

}
```

این مدل نیز HTTP Response نیست! Application فقط نتیجه Use Case را تولید می‌کند و Presenter آن را برای خروجی مناسب تبدیل می‌کند.

- **Outbound Port ها:** Interface هایی که Use Case برای ارتباط با بیرون (عمدتاً لایه چهارم مثل DB و سرویس های خارج از سیستم) نیاز دارد، Outbound Port هستند. پیاده سازی این Interface ها در لایه بیرونی قرار می‌گیرد و اینها، اینترفیس ابزار و Adapter ها و در واقع شابلون Repository هایی است که نیازهای کاربر را در لایه سخت افزار یا بیرون از سیستم هندل می‌کنند و نتیجه را به واسطه لایه دوم به لایه سوم یعنی کاربر می‌رساند.

:ProductRepository.hpp

```
#pragma once
#include <optional>
#include "../domain/entities/Product.hpp"
namespace application {

class ProductRepository {
public:
    virtual ~ProductRepository() = default;
    virtual std::optional<domain::Product> findById(int id) = 0;
};

}
```

OrderRepository.hpp

```
#pragma once
#include "../domain/entities/Order.hpp"

namespace application {

class OrderRepository {
public:
    virtual ~OrderRepository() = default;
    virtual int nextId() = 0;
    virtual void save(const domain::Order& order) = 0;
};

}
```

PaymentGateway.hpp

```
#pragma once
#include "../domain/value_objects/Money.hpp"

namespace application {

class PaymentGateway {
public:
    virtual ~PaymentGateway() = default;
    virtual bool charge(int customerId, const domain::Money& amount) = 0;
};

}
```

EmailSender.hpp

```
#pragma once
#include <string>

namespace application {

class EmailSender {
public:
    virtual ~EmailSender() = default;
    virtual void sendOrderConfirmation(int customerId, int orderId) = 0;
};

}
```

پیاده‌سازی Use Case در لایه 2: دقت کنید که ما در Use Case مجاز به در اختیار گرفتن چندین Repository در جهت رفع نیازهای کاربر هستیم!

:CreateOrderService.hpp

```
#pragma once
#include "../domain/exceptions/DomainException.hpp"
#include "../ports/inbound/CreateOrderUseCase.hpp"
#include "../ports/outbound/ProductRepository.hpp"
#include "../ports/outbound/OrderRepository.hpp"
#include "../ports/outbound/PaymentGateway.hpp"
#include "../ports/outbound/EmailSender.hpp"
```

```

namespace application {

class CreateOrderService : public CreateOrderUseCase {
    ProductRepository& products_;
    OrderRepository& orders_;
    PaymentGateway& payment_;
    EmailSender& email_;

public:
    CreateOrderService(ProductRepository& products, OrderRepository& orders,
        PaymentGateway& payment, EmailSender& email)
        : products_(products), orders_(orders),
          payment_(payment), email_(email) {}

    CreateOrderResponse execute(const CreateOrderRequest& request) override {
        domain::Order order(orders_.nextId(), request.customerId);

        for (const auto& [productId, quantity] : request.products) {
            auto product = products_.findById(productId);
            if (!product) throw domain::DomainException("Product not found");
            order.addLine(*product, quantity);
        }

        order.confirm();

        if (!payment_.charge(request.customerId, order.total()))
            throw domain::DomainException("Payment failed");

        order.markAsPaid();
        orders_.save(order);
        email_.sendOrderConfirmation(request.customerId, order.id());

        return {order.id(), order.total().cents(), true};
    }
};
}

```

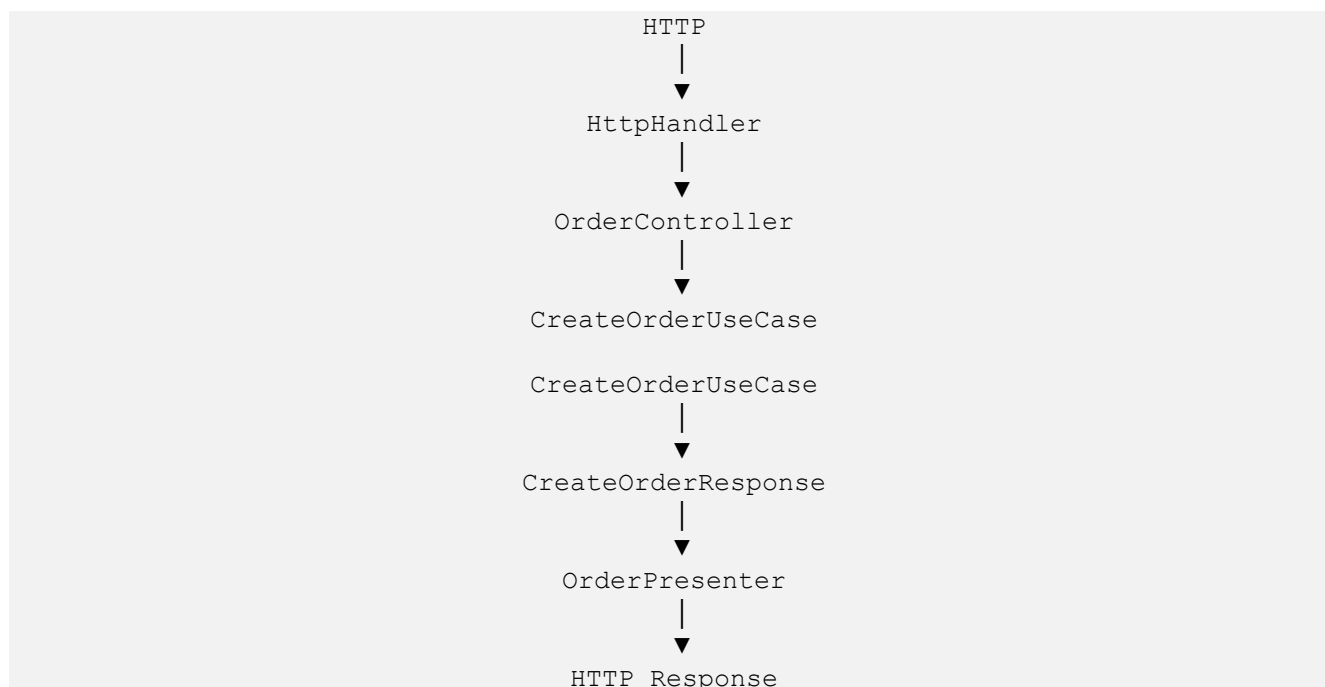
در اینجا یک Use Case به چهار Outbound Port وابسته است:

EmailSender و ProductRepository و OrderRepository و PaymentGateway

نکته مهم این است که هیچ کدام implementation واقعی نیستند و Use Case فقط Contract (Interface) مورد نیاز خود را می شناسد.

۳. لایه Interface Adapters: این لایه مسئول تبدیل مدل های دنیای بیرون (کاربران) به مدل های Application و تبدیل نتیجه

Application به مدل مناسب برای دنیای بیرون (کاربران) است و Controller و Presenter و Gateway های مربوط به Interface Adapter در این لایه قرار می گیرند.



در `HttpHandler.hpp` لایه سوم:

```

#pragma once
#include "../controllers/OrderController.hpp"

namespace adapters {

struct HttpRequest {
    int customerId;

    std::vector<std::pair<int, int>> products; //productid -> quantity
};

struct HttpResponse {
    int statusCode;

    std::string body;
};

class HttpHandler {
    OrderController& controller_;

public:
    explicit HttpHandler(OrderController& controller) : controller_(controller) {}

    HttpResponse postCreateOrder(const HttpRequest& request) {
        return controller_.createOrder(request);
    }
};

}

```


OrderController.hpp در لایه سوم:

```
#pragma once
#include "../application/ports/inbound/CreateOrderUseCase.hpp"
#include "../presenters/OrderPresenter.hpp"
#include "../gateways/HttpHandler.hpp"

namespace adapters {

class OrderController {
    application::CreateOrderUseCase& createOrder_;

public:
    explicit OrderController(application::CreateOrderUseCase& useCase)
        : createOrder_(useCase) {}

    HttpResponse createOrder(const HttpRequest& request) {
        application::CreateOrderRequest input{request.customerId, request.products};
        auto output = createOrder_.execute(input);
        return OrderPresenter::present(output);
    }
};

}
```

OrderPresenter.hpp در لایه سوم:

```
#pragma once
#include "../application/models/CreateOrderResponse.hpp"

namespace adapters {

struct HttpResponse;

class OrderPresenter {
public:
    static HttpResponse present(const application::CreateOrderResponse& response);
};

}
```

Presenter مسئول تبدیل Response مدل Application به خروجی مناسب برای Interface خارجی سمت کلاینت است. در یک پروژه واقعی می‌توان از JSON library یا امکانات Framework HTTP برای ساخت Response استفاده کرد.

۴. لایه Frameworks & Drivers؛ Infrastructure: این بیرونی‌ترین لایه است و جزئیات تکنولوژیک سیستم در آن قرار می‌گیرد؛ برای مثال Database، HTTP Framework، Stripe، SMTP و implementation‌های Repository! این لایه Outbound Port‌های تعریف‌شده در Application را پیاده‌سازی می‌کند.

InMemoryProductRepository.hpp

```
#pragma once
#include "../application/ports/outbound/ProductRepository.hpp"
#include <unordered_map>
```

```

namespace infrastructure {

class InMemoryProductRepository : public application::ProductRepository {
    std::unordered_map<int, domain::Product> products_;

public:
    InMemoryProductRepository() {
        products_.emplace(1, domain::Product(1, "Keyboard", domain::Money(5000)));
        products_.emplace(2, domain::Product(2, "Mouse", domain::Money(3000)));
        products_.emplace(3, domain::Product(3, "Monitor", domain::Money(25000)));
    }
    std::optional<domain::Product> findById(int id) override {
        auto it = products_.find(id);
        if (it == products_.end()) return std::nullopt;
        return it->second;
    }
};

}

```

InMemoryOrderRepository.hpp

```

#pragma once
#include "../application/ports/outbound/OrderRepository.hpp"
#include <unordered_map>

namespace infrastructure {

class InMemoryOrderRepository : public application::OrderRepository {
    int nextId_ = 1;
    std::unordered_map<int, domain::Order> orders_;

public:
    int nextId() override {
        return nextId_++;
    }

    void save(const domain::Order& order) override {
        orders_[order.id()] = order;
    }
};

}

```

PaymentManager.hpp

```

#pragma once
#include "../application/ports/outbound/PaymentGateway.hpp"
#include <iostream>

namespace infrastructure {

class PaymentManager : public application::PaymentGateway {
public:
    bool charge(int customerId, const domain::Money& amount) override {
        std::cout << "Charging customer " << customerId
                    << ": " << amount.cents() << " cents\n";
        return true;
    }
};

}

```

```
#pragma once
#include "../application/ports/outbound/EmailSender.hpp"
#include <iostream>

namespace infrastructure {

class EmailSenderImpl : public application::EmailSender {
public:
    void sendOrderConfirmation(int customerId, int orderId) override {
        std::cout << "Sending order confirmation for customer "
                    << customerId << ", order " << orderId << '\n';
    }
};

}
```

main.cpp:Composition Root

Composition Root محلی است که اجزای سیستم به یکدیگر متصل می‌شوند. در این نقطه Infrastructure implementation ها به Interface های Application تزریق می‌شوند و سپس Controller و Handler ساخته می‌شوند.

```
#include "../application/use_cases/CreateOrderService.hpp"
#include "../adapters/controllers/OrderController.hpp"
#include "../adapters/gateways/HttpHandler.hpp"
#include "../infrastructure/repositories/InMemoryProductRepository.hpp"
#include "../infrastructure/repositories/InMemoryOrderRepository.hpp"
#include "../infrastructure/payment/PaymentManager.hpp"
#include "../infrastructure/email/EmailSenderImpl.hpp"
#include <iostream>

int main() {
    // Layer 4: Frameworks & Drivers
    infrastructure::InMemoryProductRepository productRepository;
    infrastructure::InMemoryOrderRepository orderRepository;
    infrastructure::PaymentManager paymentManager;
    infrastructure::EmailSenderImpl emailSender;
    // Layer 2: Application Business Rules
    application::CreateOrderService createOrder(
        productRepository, orderRepository, paymentManager, emailSender);
    // Layer 3: Interface Adapters
    adapters::OrderController orderController(createOrder);
    adapters::HttpHandler httpHandler(orderController);

    adapters::HttpRequest request{100, {{1, 2}, {2, 1}}}; //{ProductId -> Quantity}

    auto response = httpHandler.postCreateOrder(request);

    std::cout << response.statusCode << '\n';
    std::cout << response.body << '\n';
}
```

جریان کامل درخواست: فرض کنیم کاربر درخواست زیر را ارسال می‌کند:

```
POST /orders

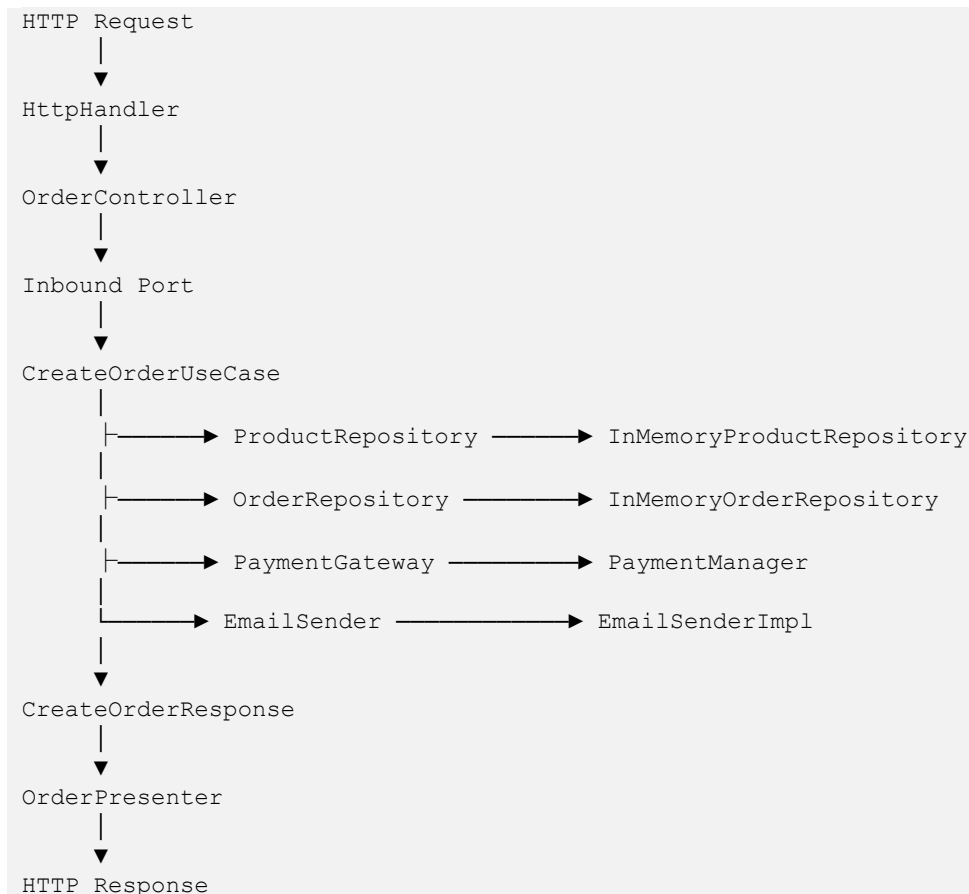
{
    "customerId": 100,
```

```

"products": [
  { "id": 1, "quantity": 2 },
  { "id": 2, "quantity": 1 }
]

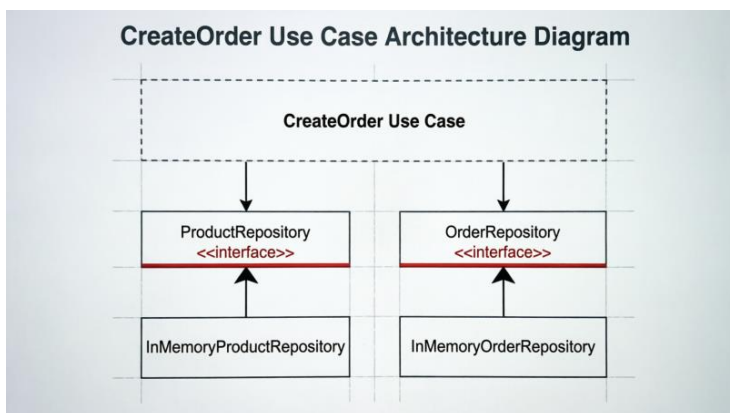
```

جریان پردازش درخواست از بیرون به داخل و سپس از داخل به بیرون به این صورت است:



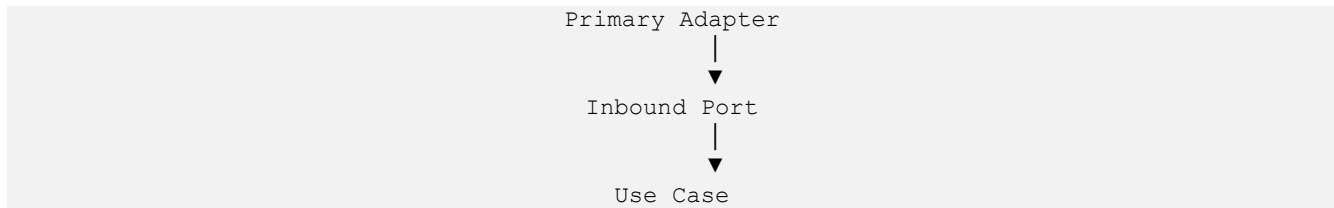
چرا Repository ها داخل Aggregate نیستند؟

Aggregate Root نباید Repository را بشناسد. اگر Order برای ساخت یا ذخیره سازی خود مستقیماً Repository داشته باشد، Domain به جزئیات Application یا Infrastructure وابسته می شود. در عوض Use Case می تواند چند Repository را همزمان دریافت کند و عملیات مربوط به Aggregate را با استفاده از آن ها orchestrate کند.



آیا Use Case همان Primary Adapter است؟

خیر. این دو نقش کاملاً متفاوتی دارند! Primary Adapter نقطه ورود به سیستم است و درخواست دنیای بیرون را به شکل قابل فهم برای Application تبدیل می‌کند ولی Use Case منطق کاربردی سیستم را مشخص می‌کند.



در این مثال HttpHandler و OrderController در سمت Primary/Input Adapter قرار می‌گیرند، CreateOrderUseCase یک Inbound Port است و CreateOrderService پیاده‌سازی Use Case را انجام می‌دهد.

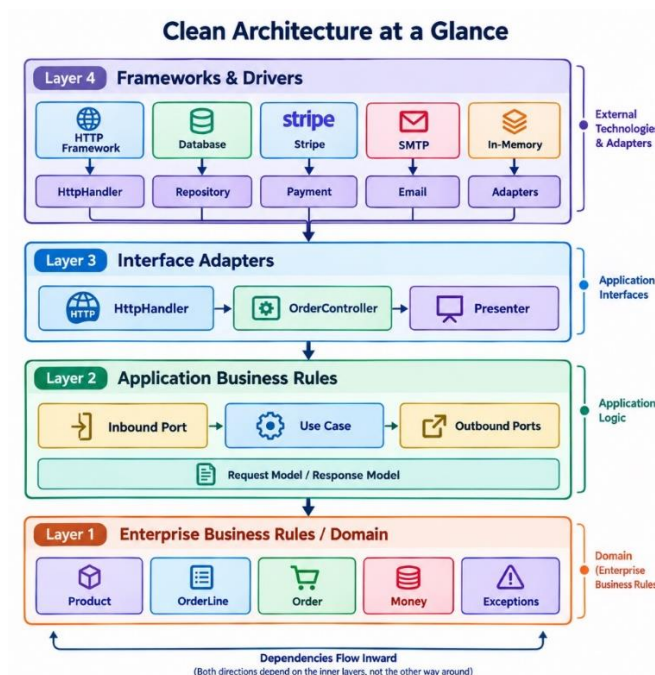
تفکیک نقش اجزای مثال:

Domain شامل Product، OrderLine، Order به‌عنوان Aggregate Root، Money به‌عنوان Value Object و DomainException است! Application شامل Use Case ها، Inbound Port ها، Outbound Port ها و Request/Response Model های کاربردی است. Interface Adapters شامل HttpHandler، Controller و Presenter است! Infrastructure شامل implementation های Repository، Payment Manager و Email Sender است.

Inbound و Outbound Port ها

Inbound Port مشخص می‌کند سیستم چگونه می‌تواند یک Use Case را اجرا کند و Outbound Port مشخص می‌کند Use Case برای انجام کار خود چه سرویس یا داده‌ای را از دنیای بیرون نیاز دارد.

Clean Architecture در یک نگاه:



HttpHandler	-> Gateway / Primary Adapter
OrderController	-> Primary / Input Adapter
CreateOrderUseCase	-> Inbound Port
CreateOrderService	-> Use Case Implementation
ProductRepository	-> Outbound Port
OrderRepository	-> Outbound Port
PaymentGateway	-> Outbound Port
EmailSender	-> Outbound Port
InMemoryProductRepository	-> Secondary / Output Adapter
InMemoryOrderRepository	-> Secondary / Output Adapter
PaymentManager	-> Secondary / Output Adapter
EmailSenderImpl	-> Secondary / Output Adapter
OrderPresenter	-> Output / Interface Adapter
Product	-> Domain Entity
OrderLine	-> Domain Entity
Order	-> Aggregate Root
Money	-> Value Object
DomainException	-> Domain Exception

پس به صورت کلی Domain اصلاً نمی‌داند HTTP، Database، Stripe یا SMTP چیست! Application نیز فقط Contract های مورد نیاز خود را می‌شناسد و به implementation های بیرونی وابسته نیست! Interface Adapters مسئول ترجمه بین مدل‌های بیرونی و مدل‌های Application هستند و Infrastructure implementation واقعی Port ها را فراهم می‌کند.

 **مثال مفهومی نهایی Clean Architecture:** برای یک مثال ساده از Clean Architecture به عنوان مثال نهایی، فرض کنید یک ساندویچ‌فروشی داریم که در آن، یک منشی مسئول دریافت سفارش مشتریان است. منشی درخواست مشتری را دریافت می‌کند و آن را به بخش اصلی سیستم، یعنی Use Case مربوط به ثبت سفارش، منتقل می‌کند. در این بخش قوانین اصلی کسب‌وکار مثل ایجاد سفارش، محاسبه قیمت و بررسی اعتبار سفارش انجام می‌شود.

در مرکز این معماری، Entity ها و Business Rule ها قرار دارند؛ مثلاً Order و Sandwich و قوانین مربوط به آن‌ها. این بخش نباید به منشی، آشپزخانه، Database یا دستگاه کارت‌خوان وابسته باشد.

 منشی را می‌توان به عنوان بخشی از Interface Adapters در نظر گرفت که درخواست دنیای بیرون را به فرم مناسب برای Use Case تبدیل می‌کند و نتیجه را نیز برای نمایش به مشتری آماده می‌کند.

از طرف دیگر، مثلاً Database، دستگاه کارت‌خوان، پرینتر سفارش، سیستم آشپزخانه و ... جزئیات بیرونی سیستم هستند که در لایه‌های خارجی قرار می‌گیرند. اگر مثلاً Database را از MySQL به PostgreSQL تغییر دهیم، نباید مجبور شویم Business Rule های Order یا Use Case های اصلی را تغییر دهیم.

در نتیجه، ایده‌ی اصلی Clean Architecture این است که وابستگی‌ها به سمت مرکز سیستم و Business Rules باشند و هسته‌ی اصلی برنامه تا حد ممکن از جزئیات تکنولوژیک و Framework ها مستقل باقی بماند.

پس در این مثال، منشی و Database و آشپزخانه قابل تعویض هستند؛ اما منطق اصلی کسب‌وکار باید مستقل و در مرکز باقی بماند.

🔧 تست نویسی در معماری های DDDمحور مطرح شده:

به جهت وابسته نبودن اصل Business به پیاده سازی فیزیکی و اصلی، به سادگی می توانیم برای بخش های مختلف این پروژه تست بنویسیم و هر بخش را به صورت جداگانه تست نماییم! فرض کنید می خواهید سناریوی پرداخت موفق و ناموفق را با سرویس پرداخت Mock یا سرویس پرداخت Fake (بدون نیاز به سرویس پرداخت واقعی که نیازمند اینترنت و زمان و در دست داشتن سرویس پرداخت خارج از سیستم و دردسرهای کم شدن پول و ... دارد) تست کنید!

```
void testCreateOrder_Success() {
    InMemoryProductRepository productRepo;
    InMemoryOrderRepositoryStub orderRepo;    // پیاده سازی ساده مشابه بالا
    FakeEmailSender emailSender;
    FakePaymentGateway payment(/*shouldSucceed=true*/);
    SimpleIdGenerator idGen;

    productRepo.save({ "P1", "Laptop", 15000000.0, 5 });
    CreateOrderUseCase useCase(productRepo, orderRepo, emailSender, payment, idGen);
    auto result = useCase.execute({ "CUST_1", { { "P1", 2 } } });
    assert(result.success);
    assert(result.totalAmount == 30000000.0);
    assert(!emailSender.sentEmails.empty());
    assert(productRepo.findById("P1")->stock == 3);    // موجودی کم شده
}

void testCreateOrder_PaymentFailed() {
    InMemoryProductRepository productRepo;
    InMemoryOrderRepositoryStub orderRepo;
    FakeEmailSender emailSender;
    FakePaymentGateway payment(/*shouldSucceed=false*/);    // پرداخت عمداً شکست می خورد
    SimpleIdGenerator idGen;

    productRepo.save({ "P1", "Laptop", 15000000.0, 5 });
    CreateOrderUseCase useCase(productRepo, orderRepo, emailSender, payment, idGen);
    auto result = useCase.execute({ "CUST_1", { { "P1", 1 } } });
    assert(!result.success);
    assert(emailSender.sentEmails.empty());    // نباید ایمیلی ارسال شده باشد
}
```

حتی می توانید تست سرویس را که روی دیتابیس MySQL یا PostgreSQL پیاده می شود را با تعویض Secondary Repository دیتابیس روی نوع In Memory به سادگی تست کنید!

📌 جمع بندی کلی هر سه معماری

هدف از معرفی این سه معماری، حفظ کردن آن‌ها به عنوان سه معماری کاملاً جدا نیست؛ هدف این است که نقش‌ها، مسئولیت‌ها و جهت وابستگی‌ها را بشناسید و بدانید هر چیز در کجای سیستم قرار می گیرد.

در عمل می توانیم از ترکیب ایده‌های هر سه، ساختاری نزدیک به Clean Architecture داشته باشیم!

HTTP Handler درخواست را از دنیای بیرون دریافت می کند و آن را به Controller مربوط به آن درخواست (OrderController، UserController، ProvinceController، CategoriesController، ProductsController) می دهد! Controller به عنوان Primary Adapter، داده ورودی را به مدل مورد نیاز Inbound Port تبدیل کرده و Use Case را اجرا می کند.

Use Case با تکیه بر Domain شامل Entity ها، Value Object ها و قواعد کسب و کار، منطق برنامه را اجرا می کند و برای دسترسی به دنیای بیرون فقط از طریق Outbound Port ها درخواست می فرستد. پیاده سازی این Port ها در لایه های بیرونی و در قالب Secondary Adapter ها قرار دارد؛ مانند Repository، Payment و Email Adapter!

نتیجه ی Use Case از طریق مدل خروجی به لایه ی Adapter برمی گردد و Presenter آن را به فرمت مناسب دنیای بیرون، مانند JSON/HTTP Response، تبدیل می کند.

توصیه بزرگان این شاخه از معماری نرم افزار این است که به جای حفظ کردن شماره و نام لایه ها، نقش‌ها و جهت وابستگی‌ها را بشناسید و بدانید دنیای بیرون به داخل وابسته است و داخل به جزئیات بیرون وابسته نیست.

Domain مرکز سیستم و محل Business Rules است؛ شامل Entity ها، Value Object ها و قواعد و Exception های مرتبط با Domain است. این‌ها الزاماً انعکاس مستقیم Table های دیتابیس نیستند؛ مدل Domain باید بر اساس Business شکل بگیرد، نه ساختار دیتابیس.

Application محل اجرای Use Case هاست؛ جایی که جریان اجرای Business را orchestrate می کند و برای ارتباط با دنیای بیرون، Port های ورودی و خروجی را تعریف می کند.

در نهایت، Framework ها، Database و سرویس های خارجی جزئیات قابل تعویض سیستم اند، نه مرکز آن.

🤖 کدام یک را انتخاب کنیم؟

از آنجا که تفاوت اصلی این سه معماری در واژگان و تأکید است نه در ساختار زیربنایی، انتخاب معمولاً به این عوامل بستگی دارد:

اگر پروژه تان مبتنی برDDD است (Bounded Context، Aggregate)، واژگان Onion طبیعی تر است.

اگر تأکید اصلی روی تست پذیری کامل و جایگزینی آسان رابط های ورودی/خروجی است (مثلاً هم زمان REST، CLI و Message Queue)، واژگان Hexagonal با تفکیک صریح Driving/Driven شفافیت بیشتری می دهد.

اگر می خواهید یک چارچوب کلی و عمومی داشته باشید که لزوماً به DDD گره نخورده، Clean Architecture بیشترین مستندات و پذیرش صنعتی را دارد.

🚀 در عمل (بسیار مهم و کلیدی): بسیاری از تیم‌های حرفه‌ای یک ترکیب شخصی‌سازی شده می‌سازند: نام‌گذاری از Clean، تفکیک

صریح Port و Adapter از Hexagonal، و مدل‌سازی دامنه از Onion/DDDD. این یک چاشنی طلایی است! 💡

🖨️ نتیجه‌ی عملی این رویکردها نیز مشابه است: تست‌پذیری بهتر، کاهش Coupling، امکان تعویض فناوری، مستقل ماندن منطق کسب‌وکار و امکان داشتن چند ورودی یا خروجی برای یک هسته‌ی مشترک.

بنابراین اگر یکی از این سه رویکرد را به‌خوبی یاد بگیرید، یادگیری دو مورد دیگر بیشتر به معنای درک تفاوت در Terminology، تأکید و نحوه‌ی سازمان‌دهی است، نه یادگیری سه دنیای کاملاً متفاوت.

Clean Architecture، Onion Architecture و Hexagonal Architecture را می‌توان سه رویکرد بسیار نزدیک برای حل یک مسئله‌ی مشترک دانست! جدا کردن منطق اصلی کسب‌وکار از جزئیات و فناوری‌های بیرونی.

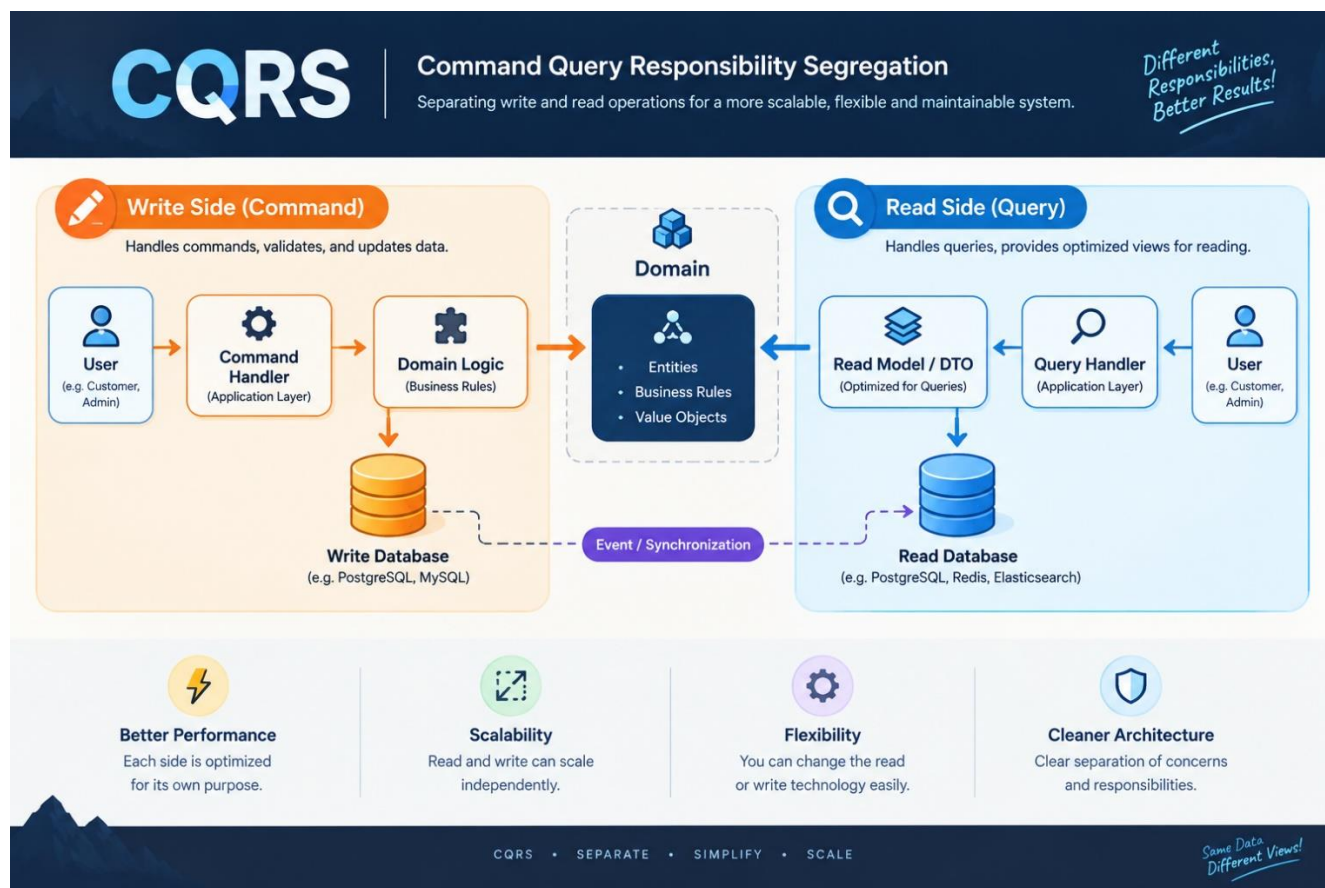
هر سه بر Dependency Inversion و جهت‌دهی وابستگی‌ها به سمت هسته‌ی سیستم تأکید دارند؛ یعنی کدهای Business نباید به جزئیات فنی مانند Database، Web Framework یا Message Broker وابسته باشند!

در هر سه رویکرد، هسته سیستم مستقل از فناوری است و جزئیات زیرساختی در لایه‌های بیرونی قرار می‌گیرند.

وابستگی‌ها باید به سمت هسته حرکت کنند؛ بنابراین مثلاً Repository Interface می‌تواند در سمت Core/Application تعریف شود و پیاده‌سازی PostgreSQL آن در Infrastructure قرار بگیرد!

هر سه معماری از تفکیک Interface از Implementation استفاده می‌کنند، اما نحوه‌ی بیان و تأکید آن‌ها متفاوت است!

- الگوی CQRS (Command Query Responsibility Segregation): نوعی الگوی معماری نرم افزار (Architectural) است که در آن، ما مکانیسم و مسئولیت Read و Write را از هم جدا می کنیم!



این الگو اولین بار توسط Greg Young حدود سال ۲۰۱۰ به صورت رسمی فرموله و نام گذاری شد، اما ریشه فکری آن به اصل قدیمی تری به نام CQS (Command Query Separation) برمی گردد که توسط Bertrand Meyer در دهه ۱۹۸۰ مطرح شده بود. این الگو می تواند در کنار DDD و Event Sourcing مکمل خوبی باشد ولی می توان مستقل از DDD هم آنرا پیاده کرد (می توان روی سورس غیر DDD هم پیاده کرد!)

🔗 فرق Command و Query

Command ها باعث ایجاد تغییر در داده ها و وضعیت سیستم می شوند، اما Query ها تنها برای خواندن اطلاعات هستند و هیچ تغییری در داده ها ایجاد نمی کنند. به همین دلیل، هنگام ارسال یک Command معمولاً به مکانیزم هایی برای اعتبارسنجی (Validation) و کنترل صحت عملیات نیاز داریم، زیرا قرار است وضعیت سیستم تغییر کند.

در معماری CQRS، با استفاده از Event Sourcing می توان تمام رویدادهایی را که باعث تغییر وضعیت سیستم شده اند، ثبت و نگهداری کرد. به این ترتیب، همواره می توان مسیر تغییرات را دنبال کرد و مشاهده نمود که سیستم چگونه به وضعیت فعلی خود رسیده است. نمونه های شناخته شده ای این رویکرد را می توان در سیستم های بانکی یا بلاک چین مشاهده کرد؛ جایی که تمامی تراکنش ها و رویدادهای مربوط به یک حساب یا کیف پول به صورت دائمی ثبت می شوند و با دنبال کردن واریز و برداشت ها می توان به مقدار و دلایل مقدار موجودی فعلی شماره حساب رسید.

در سیستم های CQRS، Command هایی که باعث تغییر وضعیت سیستم می شوند توسط **Command Service** مدیریت می شوند و رویدادهای حاصل از اجرای این Command ها نیز در **Event Sourcing** ثبت و نگهداری می شوند و تغییرات در پایگاه داده‌ی بخش **Write** اعمال می گردند. در مقابل، درخواست های خواندن اطلاعات به صورت **Query** از بخش **Read** ارسال می شوند و **Query Service** داده های مورد نیاز را از پایگاه داده‌ی بخش **Read** دریافت کرده و در اختیار رابط کاربری (UI) قرار می دهد.

CQRS چیست؟ CQRS مخفف Command Query Responsibility Segregation است، یعنی:

 تفکیک مسئولیت بین دستورها (Commands) و پرس و جوها (Queries)

 در مایکروسرویس ها CQRS می گوید:

- مدل نوشتن داده ها (Write Model) و
- مدل خواندن داده ها (Read Model)
- باید جدا از هم طراحی و پیاده سازی بشوند.

 چرا؟ مزایای این جداسازی:

مزیت	توضیح
 مقیاس پذیری بهتر	می توانیم قسمت خواندن را جدای از قسمت نوشتن scale کنیم!
 پیچیدگی کمتر	هر بخش (خواندن/نوشتن) فقط مسئول یک کار است!
 سرعت بیشتر در خواندن	می توانیم Read Model را بهینه برای کوئری سازی بسازیم (مثلاً cache شده، denormalized و ...)
 امنیت و کنترل	در Write Model می توانیم اعتبارسنجی، احراز هویت، لاگ، و event ثبت کنیم!

 مثالی از CQRS در یک سیستم سفارش:

Write Model:

CreateOrderCommand → ساخت سفارش

CancelOrderCommand → لغو سفارش

این دستورات معمولاً منجر به تغییر وضعیت سیستم و ذخیره سازی می شوند. در این بخش ممکن است از Event Sourcing استفاده شود.

Read Model:

GetOrderDetailsQuery

GetCustomerOrdersQuery

فقط برای خواندن اطلاعات از دیتابیس‌های بهینه شده (مثل Elasticsearch، Redis یا حتی View های SQL) مورد استفاده قرار می‌گیرد!

🔗 ارتباط بین Read Model و Write Model به چه شکل است؟

1- دستور اجرا می‌شود و داده‌ها (یا Event ها) ذخیره می‌شوند!

2- سرویسی (مثلاً با Kafka یا RabbitMQ یا ... یا هر پیام‌رسان Event) رویداد را پس از انجام، منتشر می‌کند! همچنان که می‌دانید، رویداد یا Event حالت پسین دارد یعنی پس از انجام یافتن کامل یک کار، رویدادش منتشر می‌شود (مثل OrderCreated یا OrderCanceled) برای مدیریت این رویدادها و ارسالشان به بخش‌های مرتبط هم می‌توان از الگوی Out Box / In Box استفاده کرد! (نیاز به مطالعه بیشتر)

3- Read Model از این رویداد مطلع می‌شود و View خود رو آپدیت می‌کند.

🔑 **سیستم‌های CRUD بیس:** سیستم‌هایی که 4 عمل اصلی Create و Read و Update و Delete را در جداول DataBase مدیریت می‌کنند! یعنی همان 4 عمل اصلی که در یک table از دیتابیس انجام می‌دهید! ابتدا یک رکورد Insert می‌کنید! سپس یا آنرا Select و یا Update و یا Delete می‌کنید!

🔍 تفاوت CQRS با CRUD (create-read-update-delete) معمولی:

ویژگی	CRUD معمولی	CQRS
همه چیز در یک مدل	✓	✗
خواندن/نوشتن ترکیب شده	✓	✗
مناسب برای سیستم‌های ساده	✓	✗
مناسب برای میکروسرویس	✗	✓
پشتیبانی از Event Sourcing	✗	✓

در ادامه یک مثال ساده از پیاده‌سازی CQRS (Command Query Responsibility Segregation) در C++ را مشاهده خواهیم کرد!

🎯 سناریو ساده: یک سیستم سفارش (Order) داریم:

- با دستور CreateOrderCommand سفارش جدید ثبت می‌کنیم (Write Model)
- با کوئری GetOrdersQuery لیست سفارش‌ها رو می‌گیریم (Read Model)

- └─ ICommand (رابط دستورات)
- └─ IQuery (رابط کوئری‌ها)
- └─ OrderWriteModel (ذخیره‌سازی سفارش‌ها)
- └─ OrderReadModel (خواندن سفارش‌ها)
- └─ CommandHandler (پردازش دستورات)
- └─ QueryHandler (پردازش کوئری‌ها)
- └─ main.cpp

 کد ساده CQRS در C++: 

```
#include <iostream>
#include <vector>
#include <string>
#include <memory>
// ----- Shared -----
struct Order {
    int id;
    std::string item;
};
// ----- Write Model -----
class OrderWriteModel {
    std::vector<Order> orders;
    int nextId = 1;
public:
    int createOrder(const std::string& item) {
        orders.push_back({nextId, item});
        return nextId++;
    }
    const std::vector<Order>& getInternalOrders() const { return orders; }
};
// ----- Read Model -----
class OrderReadModel {
    std::vector<Order> orders;
public:
    void syncFromWriteModel(const OrderWriteModel& writeModel) {
        orders = writeModel.getInternalOrders(); // Copy for simplicity
    }

    void printOrders() const {
        for (const auto& order : orders)
            std::cout << "Order #" << order.id << ": " << order.item << "\n";
    }
};
```

```
// ----- Commands -----
struct ICommand {
    virtual void execute(OrderWriteModel& model) = 0;
    virtual ~ICommand() = default;
};

class CreateOrderCommand : public ICommand {
    std::string item;
public:
    CreateOrderCommand(std::string item) : item(std::move(item)) {}

    void execute(OrderWriteModel& model) override {
        int id = model.createOrder(item);
        std::cout << "Order created with ID: " << id << "\n";
    }
};

// ----- Queries -----
struct IQuery {
    virtual void execute(const OrderReadModel& model) = 0;
    virtual ~IQuery() = default;
};

class GetOrdersQuery : public IQuery {
public:
    void execute(const OrderReadModel& model) override {
        std::cout << "All Orders:\n";
        model.printOrders();
    }
};

// ----- Main -----
int main() {
    OrderWriteModel writeModel;
    OrderReadModel readModel;
    // (Commands) اجرای دستورات
    CreateOrderCommand cmd1("Laptop");
    CreateOrderCommand cmd2("Mouse");

    cmd1.execute(writeModel);
    cmd2.execute(writeModel);
    // سینک Read Model
    readModel.syncFromWriteModel(writeModel);
    // (Queries) اجرای کوئری‌ها
    GetOrdersQuery query;
    query.execute(readModel);

    return 0;
}
```

نکات مهم: 

- CreateOrderCommand فقط با WriteModel سرو کار دارد.
- GetOrdersQuery فقط با ReadModel کار می‌کند!
- بین این دو، یک مرحله Synchronization (sync) وجود دارد که در دنیای واقعی ممکن است با Event یا Messaging انجام شود.

از **CQS** به **CQRS**: اصل (Command Query Separation) CQS می گوید هر متد باید یا یک Command باشد (کاری انجام می دهد و وضعیت را تغییر می دهد، اما چیزی بر نمی گرداند) یا یک Query باشد (داده ای بر می گرداند، اما وضعیت را تغییر نمی دهد). هرگز نباید هر دو کار را همزمان انجام دهد. مثال نقض این اصل، متدی مانند `Pop()` در یک **Stack** است که هم آیتم بالای **Stack** را بر می گرداند (Query) و هم آن را حذف می کند (Command)، این جا دو مسئولیت با هم ترکیب شده اند.

```
class OrderService {
public:
    // نقض - CQS هم وضعیت را تغییر می دهد و هم مقدار بر می گرداند
    Order ConfirmAndReturn(const Guid& orderId) {
        auto order = repository_.get(orderId);

        order.confirm();
        repository_.save(order);

        return order;
    }
    // رعایت CQS
    void Confirm(const Guid& orderId) {
        auto order = repository_.get(orderId);

        order.confirm();

        repository_.save(order);
    }
    // رعایت CQS
    OrderDto GetOrder(const Guid& orderId) const {
        return repository_.get(orderId).toDto();
    }
private:
    OrderRepository repository_;
};
```

Greg Young این ایده را از سطح یک متد به سطح کل معماری یک سیستم ارتقا داد! به جای یک مدل واحد برای خواندن و نوشتن، دو مسیر جدا طراحی می کنیم:

- مسیر **Command** (نوشتن): مسئول دریافت درخواست های تغییر وضعیت، اعتبارسنجی قوانین کسب و کار و اعمال تغییرات.
- مسیر **Query** (خواندن): مسئول بازیابی داده صرفاً برای نمایش، بدون هیچ منطق کسب و کاری.

این جداسازی می تواند از یک تفکیک ساده در سطح کلاس ها باشد یعنی **Command Handler** های جدا از **Query handler** ها که روی یک دیتابیس کار می کنند! تا معماری کاملاً توزیع شده روی دیتابیس های مختلف!

🔑 چرا اصلاً به **CQRS** نیاز داریم؟ در معماری های سنتی، معمولاً یک مدل واحد مثلاً یک مجموعه از کلاس های **Entity** هم برای نوشتن و هم برای خواندن استفاده می شود (الگوی **Repository** همه کاره):

```
class OrderService {
public:
    void CreateOrder(CreateOrderRequest request) { /* ... */ }
    void UpdateOrder(UpdateOrderRequest request) { /* ... */ }
    Order GetOrder(Guid id) { /* ... */ }
    Std::List<Order> GetOrdersByCustomer(int customerId) { /* ... */ }
    OrderSummaryReport GetMonthlyReport() { /* ... */ }
}
```

این رویکرد در سیستم های کوچک مشکلی ندارد، اما در سیستم های پیچیده به مشکلاتی برمی خورد! چرا که نیازهای خواندن و نوشتن با هم تقارن ندارند! برای مثال نوشتن نیازمند اعتبارسنجی دقیق، قوانین کسب و کاری (Invariants) و ساختار دقیق Aggregate (طبق اصول DDD) است. در صورتیکه خواندن معمولاً نیازمند شکل های مختلف نمایشی از داده است! گاهی یک لیست ساده، گاهی یک گزارش تجمیعی پیچیده، گاهی داده ای که چند Aggregate را با هم ترکیب می کند! اجبار به استفاده از یک مدل واحد هم برای خواندن و هم برای نوشتن، باعث می شود مدل نوشتن با فیلدهای اضافی برای نمایش «کثیف» شود، یا مدل خواندن مجبور شود کل ساختار پیچیده Aggregate را به دست آورد در حالی که فقط چند فیلد لازم دارد.

نکته دیگر، مشکلات کارایی است! مسیر خواندن در بیشتر سیستم ها چند برابر بیشتر از مسیر نوشتن فراخوانی می شود (نسبت رایج خواندن به نوشتن گاهی ۱۰ به ۱ یا بیشتر است). وقتی هر دو مسیر روی یک مدل و یک پایگاه داده مشترک باشند، بهینه سازی برای یکی معمولاً به ضرر دیگری تمام می شود.

مشکل دیگر، پیچیدگی Query های گزارش گیری است! وقتی از الگوهای غنی DDD (مثل Aggregate) که مخصوص نوشتن است، برای خواندن هم استفاده می کنید، بارگذاری کامل یک Aggregate بزرگ فقط برای نمایش چند فیلد در یک لیست، کاری بسیار ناکارآمد و پرهزینه است. اینجاست که DTO معنا پیدا می کند! چرا که معمولاً در Read نیاز داریم داده را به صورتی که کاربر کلاینت لازم دارد واکنشی کنیم و آنرا نمایش دهیم ولی در Write کمتر پیش می آید که بخشی از مدل دیتا را در دیتابیس ثبت کنیم! (DTO یعنی struct ای که می تولد بخشی از تمام property های مدل اصلی Table دیتابیس را داشته باشد! لذا می توانید همان را درخواست دهید و از دیتابیس دیتاهایش را پر کنید و سپس به سمت کلاینت ارسال کنید!)

از طرف دیگر، عملیات نوشتن و ثبت اطلاعات روی دیسک معمولاً پرهزینه تر از عملیات خواندن است. بنابراین، در سیستم هایی که تعداد عملیات خواندن بسیار بیشتر از عملیات نوشتن است، نگهداری داده های مورد نیاز در RAM یا Cache Memory می تواند عملکرد سیستم را به طور چشمگیری بهبود دهد؛ زیرا دسترسی به داده از حافظه بسیار سریع تر از خواندن آن از دیسک و انجام عملیات I/O است.

در چنین شرایطی می توان داده های مورد نیاز برای خواندن را در فواصل زمانی کوتاه در Cache به روزرسانی یا Reload کرد تا پردازش های خواننده، اطلاعات را مستقیماً از RAM دریافت کنند. به این ترتیب، عملیات I/O و منابع دیسک بیشتر در اختیار پردازش هایی قرار می گیرند که واقعاً نیاز به تغییر و ثبت اطلاعات دارند.

این رویکرد به خصوص در سیستم هایی که نسبت تعداد عملیات خواندن به عملیات نوشتن بسیار زیاد است، می تواند راهکاری مؤثر برای کاهش فشار روی Storage و افزایش سرعت و مقیاس پذیری سیستم باشد.

 **راه حل CQRS : CQRS** با تفکیک این دو مسیر، اجازه می دهد هر کدام مستقل از دیگری بهینه سازی شوند!

سطوح مختلف پیاده سازی CQRS:

CQRS یک مفهوم «همه یا هیچ» نیست؛ طیفی از سادگی تا پیچیدگی دارد. شناخت این سطوح برای انتخاب صحیح در پروژه بسیار مهم است.

سطح 1: جداسازی در سطح کد (Single Database) (ساده ترین سطح)

کلاس های Command و Query جدا هستند، اما هر دو از یک پایگاه داده مشترک استفاده می کنند. مدل نوشتن ممکن است Aggregate کامل DDD باشد؛ مدل خواندن می تواند مستقیماً با SQL خام یا یک ORM سبک، داده مسطح برای نمایش برگرداند (بدون عبور از لایه Domain)


```
// ===== Command =====
class ConfirmOrderCommand {
public:
    int orderId;
};

class ConfirmOrderCommandHandler {
public:
    explicit ConfirmOrderCommandHandler(IOrderRepository& repository)
        : repository_(repository){}

    void handle(const ConfirmOrderCommand& command){
        auto order = repository_.getById(command.orderId);
        order->confirm(); // Domain logic
        repository_.saveChanges();
    }
private:
    IOrderRepository& repository_;
};

// ===== Query =====
class GetOrderSummaryQuery {
public:
    int orderId;
};

class OrderSummaryDto {
public:
    int id;
    std::string customerName;
    double total;
    std::string status;
};

class GetOrderSummaryQueryHandler {
public:
    explicit GetOrderSummaryQueryHandler(IDbConnection& connection)
        : connection_(connection){}

    OrderSummaryDto handle(const GetOrderSummaryQuery& query){
        // Raw SQL – Fast access without loading the Aggregate
        constexpr std::string_view sql = R"(
            SELECT o.Id,
                   c.Name AS CustomerName,
                   o.Total,
                   o.Status
            FROM Orders o
            JOIN Customers c ON o.CustomerId = c.Id
            WHERE o.Id = ?
        )";

        return connection_.querySingle<OrderSummaryDto>(
            sql,
            query.orderId
        );
    }
private:
    IDbConnection& connection_;
};
```

این سطح از کارآیی CQRS در اکثر پروژه ها کافی و بهینه است!

سطح ۲: مدل های خواندن اختصاصی (Read Models / Materialized Views)

در این سطح، به جای Query مستقیم روی جداول تراکنشی، جداول یا View ها و DTO های مخصوص خواندن ساخته می شوند که از قبل، داده را در قالب مورد نیاز نمایش، غیرنرمال (Denormalized) و آماده نگه می دارند.

- جدول تراکنشی نرمال شده (برای نوشتن)

Orders (Id, CustomerId, Status, CreatedAt)

OrderLines (Id, OrderId, ProductId, Quantity, UnitPrice)

Customers (Id, Name, Email)

- Read Model اختصاصی! از قبل تجمیع و غیرنرمال شده (برای خواندن سریع) به بیان بهتر همچنان که مدیریت استاندارد دیتابیس، شما را ملزم به طراحی Table های نرمال شده جدا جدا بر اساس موجودیت و روابط و ... می کند، خواندن این جداول به جهت نیاز به Join کردن جداول و ...، بهینه نمی باشد! لذا در شرایطی که خواندن سریع این جداول نقش حیاتی در سرعت بالای عملیات کلاینت ها دارد، گاهی داشتن View ها و جداول Denormalized شده و پیوسته و Join شده در جهت تسرع به دست آوردن اطلاعات خواندنی لازم و حیاتی است! در اینجا جز مرحله اول که Command ها و Query ها را از هم جدا کرده ایم، نیاز داریم در بخش خواندن اطلاعات، از View ها و جداول Join شده استفاده کنیم!

OrderSummaryReadModel (OrderId, CustomerName, TotalAmount, ItemCount, StatusDisplay, LastUpdated)

این Read Model می تواند در همان پایگاه داده به صورت یک View یا جدول جداگانه (یا در پایگاه داده کاملاً متفاوتی) مثلاً Elasticsearch برای جستوجو، Redis برای کش سریع نگهداری شود. به روزرسانی این Read Model معمولاً از طریق Domain Event ها انجام می شود!

🎮 مثال: فرض کنید ما این سه جدول را مرتبط با یک Order داریم:

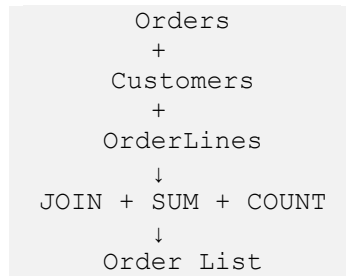
Customers

Id
Name
Orders

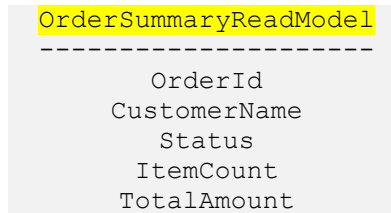
Id
CustomerId
Status
OrderLines

Id
OrderId
Quantity
UnitPrice

برای نمایش لیست سفارشات، مرتب باید این سه جدول را JOIN کنیم و جمع بزنیم:



اما برای Read Model می‌توانیم نتیجه‌ی موردنیاز را از قبل به‌صورت Denormalized نگه داریم:

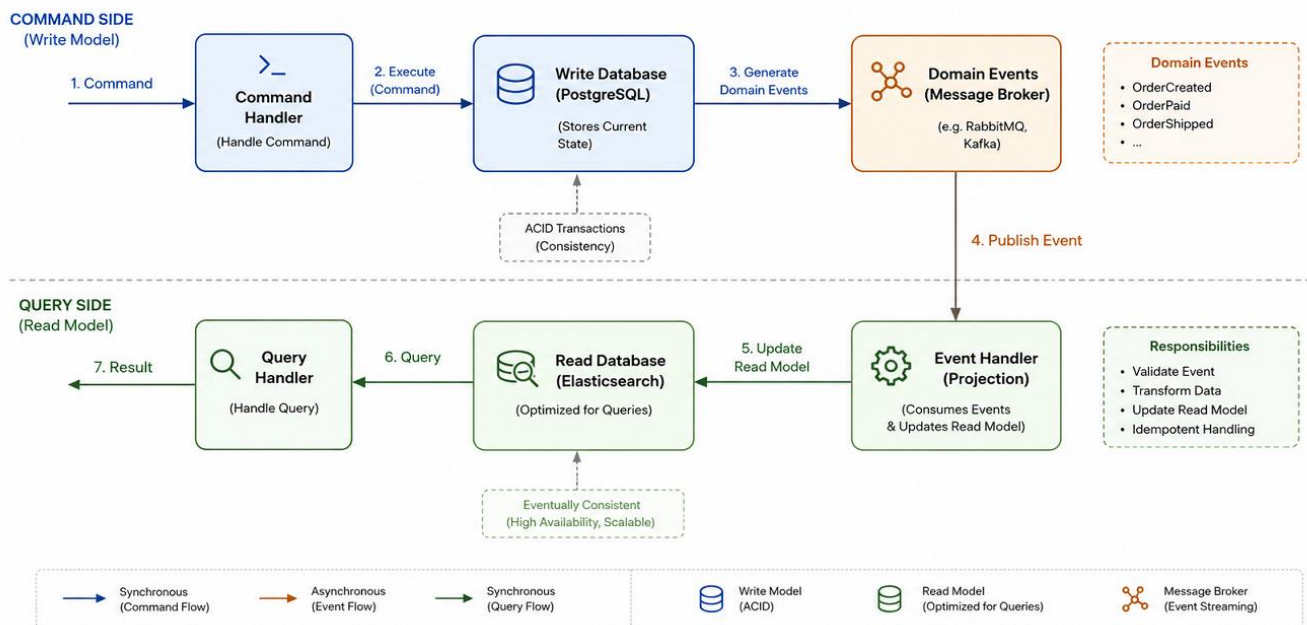


سطح ۳: پایگاه‌داده‌های کاملاً جدا (Separate Read/Write Databases)

این سطح، پیچیده‌ترین و تخصصی‌ترین سطح است و مسیر نوشتن روی یک پایگاه‌داده تراکنشی مثلاً PostgreSQL بهینه‌شده برای Consistency عمل می‌کند، و مسیر خواندن روی یک پایگاه‌داده کاملاً متفاوت مثلاً MongoDB برای اسناد آماده نمایش، یا Elasticsearch برای جست‌وجوی متنی یا مثلاً روی یک Cache Memory مانند Redis از نوع Not-Persist (روی Ram به جهت درگیر نکردن IO) عمل می‌کند. این دو با یک مکانیزم همگام‌سازی نامتقارن (Asynchronous Synchronization) معمولاً از طریق پیام‌رسانی (Message Broker) و Domain Event ها به‌روز نگه داشته می‌شوند.

CQRS Level 3 – Event-Driven Architecture

Command side writes -> Domain Events -> Projections update Read side -> Queries



❁ پیاده‌سازی کامل با الگوی (Mediator رایج‌ترین شیوه در دنیای واقعی)

در اکوسیستم‌هایی مثل NET. رایج‌ترین شیوه پیاده‌سازی CQRS استفاده از الگوی Mediator است که Command ها و Query ها را به‌عنوان اشیای جدا مدل کرده و آن‌ها را به Handler مربوطه هدایت می‌کند.

💡 **توجه:** ما در بخش "پیاده‌سازی الگوی Mediator به عنوان Command Bus در لایه 3 از معماری Clean" به تشریح چگونگی پیاده‌سازی یک Command Bus از نوع Mediator پرداخته بودیم و لذا از تکرار مکررات در این بخش اجتناب می‌کنیم!

💡 **نکته کلیدی این ساختار:** کنترلر دیگر هیچ منطقی ندارد! فقط Command/Query را می‌سازد و به Mediator می‌فرستد. این جداسازی باعث می‌شود Handler های Command و Query کاملاً مستقل، قابل تست و قابل تکامل جداگانه باشند.

💡 **نکته کلیدی (برای رفع سو تفاهم):** در مثالهای ReadModel و WriteModel این کتاب، همیشه لیستی از آیتم‌ها را داخل این دو نوع مدل نگه می‌داشتیم! در صورتیکه این کار را بر اساس ساختار In Memory Records انجام می‌دادیم! در صورتیکه اگر با دیتابیس طرف باشیم، نباید این لیست‌ها را داخل ReadModel و WriteModel بسازیم! بلکه به صورت مستقیم در شکل لیست از دیتابیس می‌خوانیم و همچنین در شکل SQLQuery در دیتابیس ذخیره می‌کنیم!

```
class OrderWriteModel {
public:
    void CreateOrder(
        int orderId,
        int customerId) {

        db.execute(
            "INSERT INTO Orders (Id, CustomerId, Status) "
            "VALUES (?, ?, 'Created')",
            orderId,
            customerId
        );
    }

    void ConfirmOrder(int orderId)
    {
        db.execute(
            "UPDATE Orders "
            "SET Status = 'Confirmed' "
            "WHERE Id = ?",
            orderId
        );
    }
private:
    Database db;
};

class OrderReadModel
{
public:
```

```

std::optional<OrderDto> GetById(int orderId)
{
    return db.querySingle<OrderDto>(
        "SELECT Id, CustomerId, Status "
        "FROM Orders "
        "WHERE Id = ?",
        orderId
    );
}
private:
    Database db;
};

```

✓ **CQRS همراه با Domain Events و ساخت Read Model:** یکی از قدرتمندترین ترکیب‌ها، استفاده از Domain Event

برای به‌روزرسانی Read Model ها است. این الگو گاهی Event-Carried State Transfer نامیده می‌شود.

✗ **دام‌های رایج (Anti-Patterns) در استفاده از CQRS:**

✗ **1- استفاده از CQRS در جایی که نیازی نیست:** بسیاری از تیم‌ها صرفاً به‌خاطر مد روز بودن CQRS پیچیده با پایگاه‌داده جدا و Event Sourcing، آنرا روی یک اپلیکیشن ساده CRUD پیاده می‌کنند و نتیجه‌اش پیچیدگی بی‌دلیل است.

2- اختلاط Microservice و CQRS: یک الگوی معماری در سطح یک سرویس/برنامه است، نه لزوماً یک تصمیم توزیع سیستم. می‌توان CQRS را در دل یک Monolith (نرم‌افزاری که توزیع شده و Microservice نیست) هم به‌خوبی پیاده کرد.

✓ **چه زمانی از CQRS استفاده کنیم؟**

معیار تصمیم‌گیری باید بر اساس پیچیدگی واقعی دامنه و الگوی بار سیستم باشد، نه ترند بودن الگو!

✓ **CQRS (حداقل سطح 1) مناسب است وقتی:**

- مدل نوشتن (Domain Model) پیچیده و غنی از قوانین کسب‌وکار است (مثلاً استفاده جدی از DDD)
- نیازهای نمایشی داده بسیار متنوع‌تر از ساختار مدل نوشتن هستند (چند View متفاوت، گزارش‌های تجمیعی، فیلترهای پیچیده)
- نسبت خواندن به نوشتن بالاست و کارایی مسیر خواندن اهمیت زیادی دارد.

✓ **سطوح پیشرفته‌تر CQRS (پایگاه‌داده جدا، Event Sourcing) مناسب است وقتی:**

- سیستم نیاز به مقیاس‌پذیری بسیار بالا در مسیر خواندن دارد (میلیون‌ها درخواست خواندن در برابر هزاران نوشتن).
- نیاز به Audit Trail کامل یا بازسازی وضعیت تاریخی سیستم وجود دارد (که Event Sourcing به‌طور طبیعی فراهم می‌کند)
- سیستم به‌صورت Event-Driven و توزیع‌شده (Microservice) طراحی شده و این هزینه زیرساختی، از قبل پذیرفته شده است.

✗ **بهتر است از CQRS پرهیز کرد وقتی:**

- برنامه یک CRUD ساده اداری یا داخلی است بدون منطق کسب‌وکاری پیچیده.
- تیم کوچک است و زمان/هزینه نگهداری زیرساخت اضافه توجیه ندارد.
- نیاز فوری به Consistency لحظه‌ای (Strong Consistency) در همه بخش‌های سیستم وجود دارد و Eventual Consistency قابل قبول نیست.

CQRS در اصل ایده‌ای ساده است: مسیر نوشتن و مسیر خواندن دو نیاز متفاوت دارند، پس نباید مجبور باشند از یک مدل استفاده کنند! این ایده از یک اصل ساده در سطح متد (CQS) شروع شد و توسط Greg Young به یک الگوی معماری کامل تبدیل شد که می‌تواند از یک جداسازی بسیار ساده در سطح کلاس‌ها (بدون هیچ هزینه زیرساختی) تا یک معماری کاملاً توزیع‌شده با پایگاه‌داده‌های جدا و Event Sourcing گسترش یابد.

نکته حیاتی این است که CQRS یک الگوی طیفی است، نه یک تصمیم دودویی. بهترین رویکرد معمولاً شروع از ساده‌ترین سطح (جداسازی Command/Query در کد، با یک پایگاه‌داده مشترک) است و اگر کفایت نکرد، باید به فکر سطوح عمیق‌تر مثل جدا کردن دیتابیس خواندن و نوشتن شد! دقیقاً همان فلسفه‌ای که در معماری‌های Clean، Onion و Hexagonal هم دیدیم! مرزها و انتزاعات باید متناسب با پیچیدگی واقعی مسئله باشند، نه پیش‌فرضی که همه‌جا اعمال می‌شود.

The Power of ECS

Entity + Component + System

A modern architecture for high-performance game worlds

More Entities. More Fun.

Game World

- Many entities
- Dynamic behavior
- Real-time performance

Entity

A lightweight ID that represents an object in the game world.

E001

E002

E003

E004

E005

E006

E007

E008

...
Just an ID

Component

Data that describes the entity. Small. Focused. Reusable.

Transform (position, rotation, scale)

Velocity (speed, direction)

Health (current, max)

Renderable (mesh, material)

AI (state, behavior)

...

Data, not behavior

System

Contains the logic. Operates on entities with required components.

Movement System (Transform + Velocity)

Combat System (Health + Damage)

Render System (Renderable + Transform)

AI System (AI + Transform)

...

Logic, not data

Why ECS?

Built for games. Ready for scale.

High Performance

Cache-friendly and data-oriented

Massive Scalability

Handle thousands of entities

Flexibility

Easy to add new features

Maintainability

Clean separation of concerns

SAME WORLD • THOUSANDS OF ENTITIES • ONE ARCHITECTURE

✱ **مقدمه:** چرا معماری‌های سنتی در بازی‌سازی شکست می‌خورند؟ برای سال‌ها، برنامه‌نویسی شیء‌گرا (OOP) و الگوی MVC استاندارد طلایی توسعه نرم‌افزار بودند. اما در توسعه بازی‌های سه بعدی مدرن، این الگوها به کابوس تبدیل شدند.

✗ چرا در بازیها، MVC جواب نمی‌دهد؟

MVC برای برنامه‌های Event-Driven (مثل وب و دسکتاپ) طراحی شده است، در حالی که بازی‌ها Game Loop-Driven هستند. در MVC برای بازیها، جداسازی شدید Model و View باعث ایجاد سربار (Overhead) وحشتناکی می‌شود و مدیریت ورودی‌های پیوسته (مثل آنالوگ دسته بازی) با Controller بسیار دشوار است!

✗ چرا در بازیها، OOP کلاسیک شکست می‌خورد؟

در دهه ۱۹۹۰، بازی‌های ویدیویی ساده بودند. یک بازی ممکن بود ۱۰ تا ۲۰ نوع آبجکت داشته باشد که هر کدام یک کلاس جداگانه داشتند و برای چنین سیستمی، OOP کافی بود! اما با رشد بازی‌ها:

Quake: ۱۹۹۵ → چند صد آبجکت

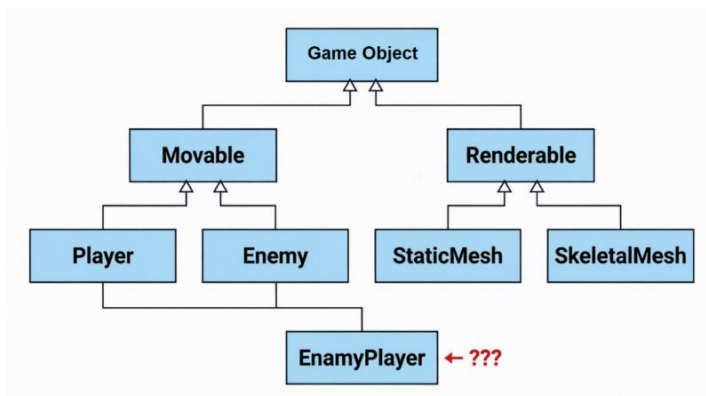
Crysis: ۲۰۰۵ → چند هزار آبجکت

Fallout: ۲۰۱۵ → ده‌ها هزار آبجکت

Starfield: ۲۰۲۳ → میلیون‌ها آبجکت

Scott Bilas در GDC 2002 اولین بار مقاله‌ای منتشر کرد به نام "A Data-Driven Game Object System" که بذر ECS را کاشت. Unity در 2018 رسماً DOTS (Data-Oriented Technology Stack) را معرفی کرد که ECS خالص بود.

1- **مشکل الماس (Diamond Problem) و ارث‌بری عمیق:** ساخت کلاس‌هایی مثل FlyingEnemy که از Enemy و FlyingObject ارث می‌برند، سلسله‌مراتب‌های شکننده و غیرقابل نگهداری ایجاد می‌کند. ما در بسیاری از سناریوهای بازی، نیاز داریم با الحاق انواع کامپوننت‌ها به اشیای مختلف، ترکیبات مختلفی از اشیاء را بسازیم و با داشتن OOP مجبوریم ارث‌بری‌های چندگانه داشته باشیم که در بسیاری از زبان‌های برنامه‌نویسی و بسیاری از مکاتب، Bad Practice شناخته می‌شود!



2- **کلاس‌های پایه متورم (Bloated Base Class):** یک کلاس GameObject که همه ویژگی‌ها (فیزیک، رندر، صدا، هوش مصنوعی) را در خود دارد، باعث می‌شود یک شیء ساده مثل Rock حافظه‌ای معادل یک کاراکتر کامل اشغال کند در صورتیکه نباید این چنین باشد!

Component / Data	 Player	 Enemy	 Rock
Position	✓	✓	✓
Mesh	✓	✓	✓
Health	✓	✓	X
AI	X	✓	X
Sound	✓	X	X

3- **Cache Miss (دشمن پنهان عملکرد):** در OOP، داده‌های یک آبجکت در حافظه RAM پراکنده (Scattered) هستند. وقتی CPU

می‌خواهد موقعیت هزاران کاراکتر را آپدیت کند، مدام دچار Cache Miss می‌شود و زمان زیادی را منتظر رم می‌ماند. فرض کنید سیستم «حرکت» (Movement System) می‌خواهد فقط position و velocity یک کاراکتر را آپدیت کند. وقتی CPU می‌خواهد position یک کاراکتر را بخواند، چون داده‌ها در یک struct بزرگ (مثلاً ۲۰۰ بایتی) پشت سر هم هستند، CPU مجبور است کل آن ۲۰۰ بایت را از رم به Cache بیاورد، در حالی که فقط به ۲۴ بایت (دو تا Vector3) نیاز دارد!

نتیجه: ۹۰٪ داده‌هایی که وارد کش CPU می‌شوند، دور ریخته می‌شوند (Cache Miss) و اتلاف شدید پهنای باند حافظه).

حالت دوم: آرایه ۵۰ تایی در ECS (نگاه DOD با Sparse-Dense)

 **راه‌حل رادیکال، ECS و طراحی داده‌محور (DOD):**

معماری ECS فقط یک الگوی طراحی نیست؛ بلکه یک تغییر پارادایم (Paradigm Shift) از شیء‌گرایی به طراحی داده‌محور (Data-Oriented Design) است.

 **کالبدشکافی سه‌گانه (Entity, Component, System)**

در ECS، ما اشیا (Objects) را فراموش می‌کنیم و همه چیز را به سه مفهوم مجزا تجزیه می‌کنیم:

1- **موجودیت (Entity):** یک Entity فقط یک شناسه یکتا (ID) است و هیچ چیز دیگری نیست (معمولاً یک عدد صحیح ۲ تا ۶۴ بیتی) است

و نقش Entity فقط مثل یک Label است و عامل تولید یک مفهوم همچون Database Record است و نقش یک پونس که چندین فیلد (کامپوننت‌های مختلف) را به هم متصل می‌کند را بازی می‌کند تا یک Record بوجود آورد.

Entity 0 → Player

Entity 1 → Enemy

Entity 2 → Rock

Entity 3 → Bullet

2- جزء / کامپوننت (Component): کامپوننت‌ها فقط و فقط داده‌های خام هستند. هیچ تابع (Method)، هیچ منطق (Logic) و هیچ

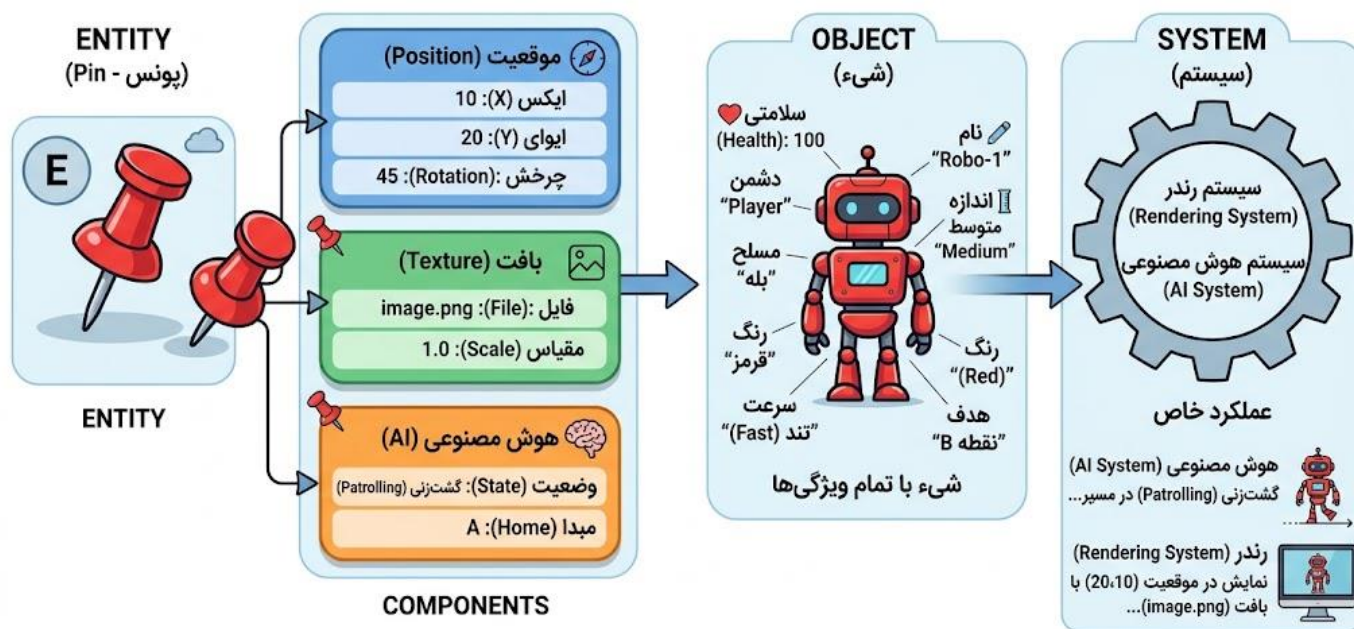
رفتاری در کامپوننت وجود ندارد. آن‌ها معمولاً به صورت struct پیاده‌سازی می‌شوند تا سربار حافظه نداشته باشند و نقش کامپوننت‌ها حالت (State) بازی را تعریف می‌کنند.

```
struct TransformComponent { float x, y, z; float rotX, rotY, rotZ; };
struct VelocityComponent { float vx, vy, vz; };
struct HealthComponent { float current, max; };
struct RenderComponent { string meshPath, material; };
```

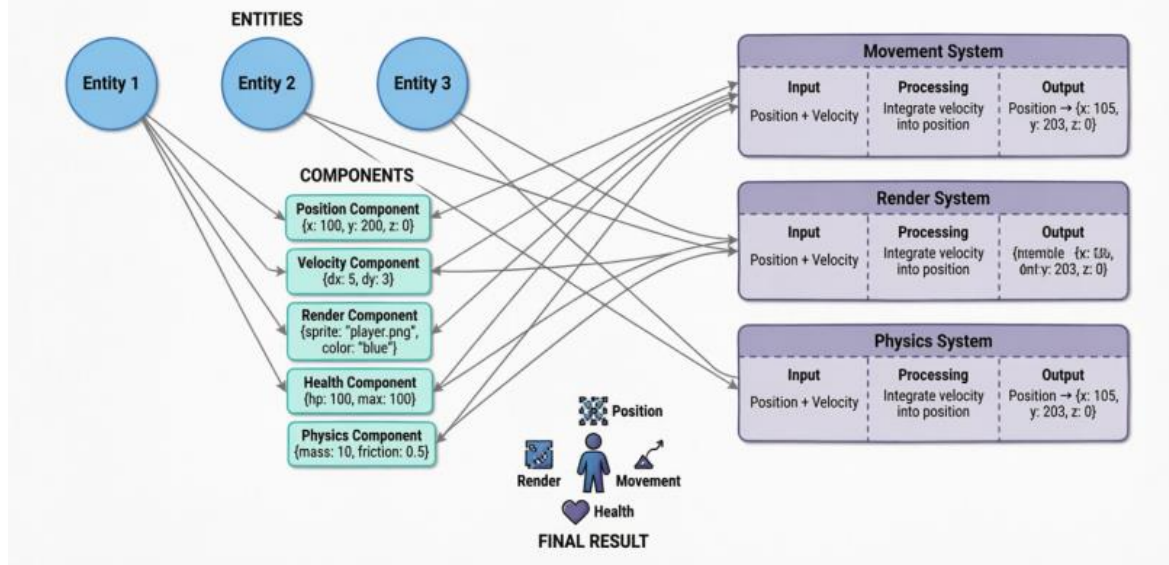
6- سیستم (System): سیستم‌ها فقط و فقط منطق و رفتار هستند. سیستم‌ها، داده‌ها را در خود نگه نمی‌دارند. وظیفه یک System این است که تمام Entityهایی را که دارای مجموعه خاصی از کامپوننت‌ها هستند را پیدا کند (Query) و روی داده‌هایشان، عملیات (اجرای تابع) انجام دهد.

```
void MovementSystem::Update(World& world, float deltaTime) {
    // پیدا کردن تمام موجودیت‌هایی که هم transform دارند و هم Velocity
    for (Entity e : world.Query<TransformComponent, VelocityComponent>()) {
        auto& pos = world.Get<TransformComponent>(e);
        auto& vel = world.Get<VelocityComponent>(e);
        pos.x += vel.vx * deltaTime;
        pos.y += vel.vy * deltaTime;
        pos.z += vel.vz * deltaTime;
    }
}
```

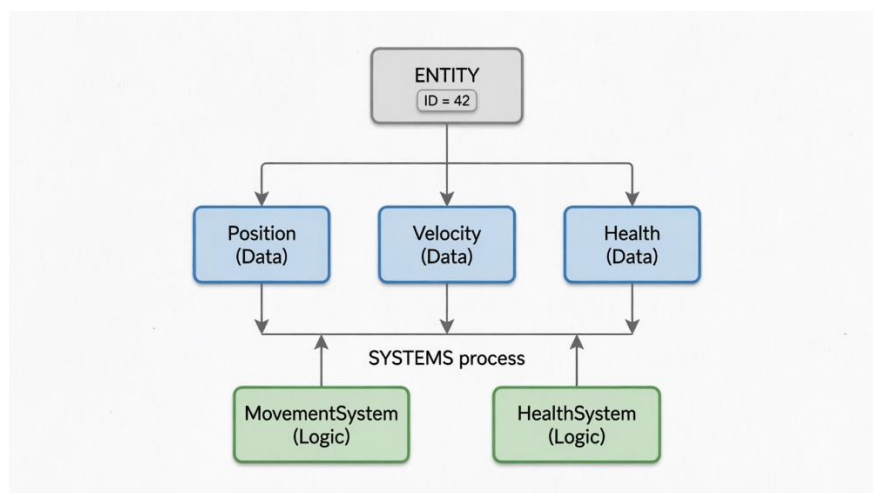
معماری کامپوننت سیستم موجودیت (ECS)



ECS (Entity-Component-System) Architecture Diagram



رابطه سه گانه:



🚀 **سلاح مخفی ECS، طراحی داده محور (DOD) و Cache Locality:** ارزش اصلی ECS در عملکرد (Performance) و مدیریت حافظه است.

❌ **مشکل (AoS - Array of Structs) در OOP:** داده های اشیاء در حافظه پراکنده اند. CPU برای پردازش هر آبجکت باید به آدرس های دور برود و Cache Line مدام پر و خالی می شود.

💡 **راه حل (SoA - Struct of Arrays) در ECS:** در ECS مدرن، داده ها بر اساس نوع کامپوننت در حافظه چیده می شوند. تمام Position ها در یک آرایه پیوسته (Contiguous Array) و تمام Velocity ها در یک آرایه پیوسته دیگر قرار می گیرند. وقتی MovementSystem اجرا می شود، CPU فقط آرایه های پیوسته را می خواند. Cache Hit به حداکثر می رسد و ECS می تواند عملکرد را ۱۰ تا ۱۰۰ برابر سریع تر از OOP سنتی کند.

🏠 معماری‌های داخلی ECS و مدیریت حافظه: برای پیاده‌سازی ECS در سطح موتورهای AAA، سه رویکرد اصلی وجود دارد:

1- **Naive ECS**: ساده اما کند است! هر Entity یک `std::unordered_map` از کامپوننت‌ها دارد.

✅ مزایا: ساده، قابل فهم.

❌ معایب: Cache Miss زیاد، Heap Allocation زیاد و کند.

```
class Entity {
private:
    std::unordered_map<
        std::type_index,
        std::shared_ptr<Component>
    > components;
};
```

2- **Archetype ECS**: (مناسب پردازش انبوه) Entity‌هایی که دقیقاً ترکیب یکسانی از کامپوننت‌ها دارند، در یک گروه به نام Archetype قرار می‌گیرند. داده‌های هر Archetype در بلوک‌های حافظه‌ای متراکم به نام Chunk ذخیره می‌شوند تا Locality حافظه و سرعت پردازش دسته‌ای افزایش یابد.

✅ مزایا: Cache Locality عالی، مناسب برای پردازش انبوه و سریع Entity‌ها.

❌ معایب: پیچیده‌تر، و افزودن یا حذف کامپوننت در زمان اجرا (Structural Change) معمولاً پرهزینه است.

Archetype A: [Transform][Health][Render]

Entity 0: [t0][h0][r0]

Entity 1: [t1][h1][r1]

Entity 5: [t5][h5][r5]

Archetype B: [Transform][Health][Render][AI]

Entity 2: [t2][h2][r2][ai2]

Entity 3: [t3][h3][r3][ai3]

Archetype C: [Transform][Render]

Entity 4: [t4][r4]

3- **Sparse Set ECS**: (مناسب ساختارهای پویا) برای هر نوع کامپوننت یک Sparse Set جداگانه داریم. این ساختار از آرایه‌های Sparse و

Dense استفاده می‌کند تا دسترسی، اضافه و حذف کامپوننت را معمولاً با هزینه $O(1)$ و Iteration سریع و Cache-Friendly فراهم کند.

✅ مزایا: اضافه/حذف کامپوننت سریع، انعطاف بالا و مناسب برای Entity‌های پویا.

🚀 مفاهیم پیشرفته و زیرساختی (Advanced ECS): برای ساخت یک موتور بازی مدرن، درک مفاهیم زیر الزامی است:

1- **Query Engine**: موتور باید بتواند Entity‌هایی را که دارای کامپوننت‌های خاص هستند (با شروط All, Any, None) با کمترین سر بار پیدا کند.

2- Command Buffer: در حین اجرای یک System، نباید Entity ها را ایجاد یا حذف کرد (چون حافظه در حال پردازش است). تغییرات ساختاری (Structural Changes) باید در یک Command Buffer ذخیره شده و در انتهای فریم (Playback) اعمال شوند.

3- Job System & Multithreading: چون داده‌ها در آرایه‌های پیوسته هستند و System ها Stateless (بدون وضعیت داخلی) می‌باشند، می‌توان آرایه‌ها را به چند بخش تقسیم کرد و به هسته‌های مختلف CPU داد تا همزمان (Parallel For) پردازش شوند.

4- Event Bus: برای ارتباط کم‌وابسته (Loose Coupling) بین System ها. مثلاً وقتی سلامت یک Entity به صفر می‌رسد، یک DeathEvent منتشر می‌کند و سیستم‌های صدا و UI بدون اینکه مستقیماً به هم متصل باشند، آن را دریافت می‌کنند. (الگوی Observer)

🌸 پیاده‌سازی عملی و کامل با C++

پیاده‌سازی Naive ECS (آموزشی و کامل): این کد یک پیاده‌سازی کامل از یک موتور بازی کوچک با استفاده از رویکرد Naive، همراه با Event Bus و Factory است.

```
#include <cmath>
#include <vector>
#include <memory>
#include <string>
#include <limits>
#include <stdint>
#include <iostream>
#include <typeindex>
#include <algorithm>
#include <functional>
#include <unordered_map>

using namespace std;
// =====
//   COMPONENTS - فقط داده
// =====

struct Component {
    virtual ~Component() = default;
};

struct TransformComponent : Component {
    float x = 0;
    float y = 0;
    float z = 0;
    float rotY = 0;
    float scale = 1;
};

struct VelocityComponent : Component {
    float vx = 0;
    float vy = 0;
    float vz = 0;
};

struct HealthComponent : Component {
    float hp = 100;
    float maxHp = 100;
};
```

```

struct RenderComponent : Component {
    string mesh;
    string material;
    bool visible = true;
};

struct AIComponent : Component {
    enum class State {
        Idle, // وضعیت عادی هوش مصنوعی
        Chase, // وضعیت مشکوک شدن و دنبال کردن چیز مشکوک
        Attack, // وضعیت اтак زدن و حمله کردن
        Flee, // وضعیت گریختن و فرار کردن
        Dead // وضعیت مردن
    };

    State state = State::Idle;
    float detectionRange = 15.0f; // محدوده تشخیص دشمن
    float attackRange = 2.0f; // محدوده اтак زدن به دشمن
    float speed = 3.0f;
};

struct TagComponent : Component {
    string tag;

    TagComponent(string t)
        : tag(move(t)) {}
};

// =====
// ENTITY
// =====

using EntityID = uint32_t;
// حد آستانه آی دی (ماکسیمم آی دی ممکن)
constexpr EntityID INVALID_ENTITY = numeric_limits<EntityID>::max();

class Entity {
    EntityID m_id;
    bool m_active = true; // آیا انتیتی اکتیو است یا خیر؟
    // تایپ ایندکس ابزاری است که برای هر نوع داده، عددی یونیک را اختصاص می دهد
    // در اینجا برای هر نوع کامپوننت، عددی یونیک تنظیم می کند
    unordered_map<
        type_index,
        shared_ptr<Component>
    > m_components;

public:
    explicit Entity(EntityID id)
        : m_id(id) {}

    EntityID id() const {
        return m_id;
    }

    bool active() const {
        return m_active;
    }

    void setActive(bool active) {
        m_active = active;
    }
};

```

```

می خواهیم یک نوع کامپوننت را با لیستی از ورودی هایش به نسبت کانستراکتور مختص
آن کامپوننت بسازیم و در مپ کامپوننت های مختص این انتیتی ملحق کنیم
فرضا این انتیتی مختص یک کاراکتر انسانی است و می خواهیم کامپوننت هوش مصنوعی
را با پارامترهای ورودی کامپوننت هوش مصنوعی داخل انتیتی بسازیم و کامپوننت هوش
را به عنوان نوع هوش در مپ کامپوننت های مختص این انتیتی قرار دهیم
به نوعی این متد، نوعی فکتوری متد است که قبلا در بخش فکتوری متد توضیح داده ایم
template<typename T, typename... Args>
T& addComponent(Args&&... args) {
    // std::forward preserves the lvalue or rvalue nature of the data, just like the original!
    auto component = make_shared<T>(std::forward<Args>(args)...);
    m_components[typeid(T)] = component;

    return *component;
}
یافتن کامپوننت نوع تی در کامپوننت های این انتیتی
template<typename T>
T* get() {
    auto it = m_components.find(typeid(T));

    if (it == m_components.end())
        return nullptr;

    return static_cast<T*>(it->second.get());
}

template<typename T>
bool has() const {
    return m_components.count(typeid(T)) > 0;
}

template<typename T>
void remove() {
    m_components.erase(typeid(T));
}
فرض کنید می خواهیم چک کنیم که آیا یک انتیتی تمام کامپوننت های زیر را دارد؟
// entity.hasAll<TransformComponent, VelocityComponent, HealthComponent>();
template<typename... Ts>
bool hasAll() const {
    return (has<Ts>() && ...);
    // اینجا تی اس یعنی TransformComponent , VelocityComponent , HealthComponent
    // (has<Ts>() && ...) معادل است با عبارت زیر
    // has<TransformComponent>() && has<VelocityComponent>() && has<HealthComponent>()
}
};
// =====
// EVENT SYSTEM
// =====
struct Event { // کلاس پایه ایونت ها یا رویدادها
    virtual ~Event() = default;
};

struct DamageEvent : Event { // رویداد ضربه خوردن یک بازیکن یا ...
    EntityID target; // آی دی انتیتی ضربه خورده
    float amount; // میزان ضربه
    string source; // منبع ضربه
};

struct DeathEvent : Event { // رویداد مرگ یک کاراکتر
    EntityID entity; // آی دی انتیتی
    string cause; // دلیل مرگ
};

```

```

class EventBus { // ایونت باس (پابلیش/ساب اسکرایبر) برای مدیریت رویدادها
    unordered_map< // یک رویداد که منتشر یا ایمت شود، کال بک بایند شده آن اجرا می شود>
        type_index,
        vector<function<void(const Event&)>>
    > m_listeners; // کلیدش تایپ آی دی نوع رویداد یا ایونت هست
    // ولیویش، آرایه ای از کال بک هاست که شی رویداد منتشر شده از ورودیش وارد کال بک می شود
    // پس با انتشار ایونت به واسطه شی ایونت باس، تمام کال بک های لیست رویداد مربوطه
    // اجرا خواهد شد و ایونت ارسال شده هم از ورودی تمام آنها وارد خواهد شد
    // این رویداد شامل تمام مشخصات مورد نیاز مثل آی دی یا تگ یا نام انتیتی مربوطه است
public:
    // متد رجیستر کردن رویداد همراه با دادن کال بک مربوطه به آن
    template<typename T>
    void on(function<void(const T&)> callback) {
        m_listeners[typeid(T)].push_back(
            [callback](const Event& event) {
                callback(static_cast<const T&>(event));
            }
        );
    }
    // تابع انتشار رویدادی خاص از بیرون از این کلاس
    template<typename T>
    void emit(const T& event) {
        auto it = m_listeners.find(typeid(T));

        if (it != m_listeners.end()) {
            for (auto& fn : it->second)
                fn(event);
        }
    }
};

// =====
// WORLD- دنیای بازیمان
// =====

class World {
    vector<shared_ptr<Entity>> m_entities; // لیست تمام انتیتی های مرحله بازی
    EntityID m_nextId = 1; // آی دی انتیتی بعدی با حالت افزایشی
    EventBus m_eventBus; // شی باس رویداد یا ایونت باس
public:
    EventBus& events() {
        return m_eventBus;
    }
    // ساخت انتیتی جدید با داشتن کامپوننت تگ به شرطی که تگ را خالی ارسال نکنید
    shared_ptr<Entity> create(const string& tag = "") {
        auto entity = make_shared<Entity>(m_nextId++);
        if (!tag.empty())
            entity->addComponent<TagComponent>(tag);

        m_entities.push_back(entity);
        return entity;
    }
    // یافتن انتیتی دارای کامپوننت تگ که تگش برابر با تگ ورودیست
    shared_ptr<Entity> findByTag(const string& tag) {
        for (auto& entity : m_entities) {
            auto* tagComponent = entity->get<TagComponent>();

            if (tagComponent &&
                tagComponent->tag == tag)
                return entity;
        }
        return nullptr;
    }
};

```



```

shared_ptr<Entity> findById(EntityID id) { //دی از آ استفاده با انتیتی یافتن
    for (auto& entity : m_entities) {
        if (entity->id() == id)
            return entity;
    }

    return nullptr;
}

//اجرای تابع کال بک ورودی روی تمام انتیتی های فعالی که شامل تمام کامپوننت های لیست تمپلیت است
template<typename... Ts>
void each(function<void(Entity&)> fn) {
    for (auto& entity : m_entities) {
        if (entity->active() &&
            entity->hasAll<Ts...>()) {
            fn(*entity);
        }
    }
}

size_t entityCount() const {
    return m_entities.size();
}
};

//=====
//      SYSTEMS
//=====

class PhysicsSystem {
public:
    void update(World& world, float dt) {
        // اینجا در جهان بازی، همه انتیتی های دارای کامپوننت ترنسفورم و ولوسیتی را پیدا می کنیم
        // چون موتور فیزیک به این نوع کامپوننت ها نیاز دارد
        world.each<
            TransformComponent,
            VelocityComponent
        >([&](Entity& entity) {
            //گرفتن کامپوننت ترنسفورم انتیتی
            auto* transform = entity.get<TransformComponent>();
            //گرفتن کامپوننت سرعت انتیتی
            auto* velocity = entity.get<VelocityComponent>();

            velocity->vy -= 9.8f * dt; // تنظیم سرعت رو به پایین شی به سمت جاذبه زمین
            //حرکت به سوی سرعت شی در جهات مختلف
            transform->x += velocity->vx * dt;
            transform->y += velocity->vy * dt;
            transform->z += velocity->vz * dt;

            if (transform->y <= 0) { // اگر اشیاء از بالا به زمین بازی رسید،
                transform->y = 0; // ارتفاع شی روی زمین تنظیم شود
                velocity->vy = 0; // سرعت شی به سمت پایین صفر شود تا سقوط نکند
            }
        });
    }
};

class AISystem {
public:
    //متد به روز رسانی هوش مصنوعی بازی
    void update(World& world, float dt) {
        auto player = world.findByTag("player"); // انتیتی پلیر ما
        if (!player)
            return;
    }
};

```

```

auto* playerTransform = player->get<TransformComponent>();
if (!playerTransform)
    return;

world.each<
    AIComponent,
    TransformComponent,
    HealthComponent
>([&](Entity& entity) {

    auto* ai = entity.get<AIComponent>();
    auto* transform = entity.get<TransformComponent>();
    auto* health = entity.get<HealthComponent>();

    if (health->hp <= 0) { // تشخیص بازیکنان هوشمند مرده
        ai->state = AIComponent::State::Dead; // تنظیم وضعیت مرده
        return;
    }
    // محاسبات فاصله انتیتی هوشمند از بازیکن ما
    float dx = playerTransform->x - transform->x;
    float dz = playerTransform->z - transform->z;
    float distance = sqrt(dx * dx + dz * dz);
    // اگر وضعیت هوشمند کاراکتر دشمن معمولی بود و ما در فاصله دیتکشن وی بودیم
    if (ai->state == AIComponent::State::Idle &&
        distance < ai->detectionRange) {
        ai->state = AIComponent::State::Chase;
        // تغییر وضعیت دشمن به دنبال کردن ما
    }
    // اگر دشمن در وضعیت دنبال ما آمدن بود
    if (ai->state == AIComponent::State::Chase) {
        float length = sqrt(dx * dx + dz * dz);
        // نزدیک شدن دشمن به ما به تدریج البته به روش ساده برای سهولت آموزش
        if (length > 0) {
            transform->x +=
                (dx / length) *
                ai->speed *
                dt;
            transform->z +=
                (dz / length) *
                ai->speed *
                dt;
        }
        // اگر فاصله دشمن ما از کاراکتر ما، کمتر از رنج فاصله اтак زدن بود
        if (distance < ai->attackRange)
            ai->state = AIComponent::State::Attack; // تغییر وضعیت دشمن به اтак
    }
    // اگر دشمن ما در وضعیت اтак بود و فاصله بین ما کمتر از فاصله اтак بود
    if (ai->state == AIComponent::State::Attack &&
        distance < ai->attackRange) {
        // ایونت باس یک رویداد ضربه از دشمن به ما منتشر می کند
        world.events().emit(
            DamageEvent{
                player->id(),
                10.0f,
                "enemy"
            }
        );
    }
});
};

```

```

class HealthSystem { // کامپوننت سلامتی کاراکترهای میرا
    bool m_listenerRegistered = false;

public:
    void init(World& world) {
        if (m_listenerRegistered)
            return;
        world.events().on<DamageEvent>(
            [&world](const DamageEvent& event) {

                auto entity = world.findById(event.target);

                if (!entity)
                    return;

                auto* health = entity->get<HealthComponent>();

                if (!health || health->hp <= 0)
                    return;

                health->hp = max( 0.0f, health->hp - event.amount);

                cout << "[HEALTH] Entity "
                     << event.target
                     << " took "
                     << event.amount
                     << " damage. HP: "
                     << health->hp
                     << "\n";

                // انتشار رویداد مرگ انتیتی میرا که سلامتش تمام شده
                if (health->hp <= 0) {
                    world.events().emit(
                        DeathEvent{
                            entity->id(),
                            event.source
                        }
                    );
                }
            }
        );

        m_listenerRegistered = true;
    }

    void update(World& world, float dt) {
        // Health behavior is event-driven.
    }
};

// کلاس رندر سیستم
class RenderSystem {
public:
    void update(World& world, float dt) {
        world.each<
            TransformComponent, // انتیتی های دارای کامپوننت ترنسفورم
            RenderComponent      // انتیتی های دارای کامپوننت رندر
        >([&](Entity& entity) {
            auto* transform = entity.get<TransformComponent>();
            auto* render = entity.get<RenderComponent>();
        });
    }
};

```

```

        if (render->visible) {
            cout << "[RENDER] Entity "
                << entity.id()
                << " at ("
                << transform->x << ", "
                << transform->y << ", "
                << transform->z << ")\n";
        }
    });
}

};
// =====
//   انجین شامل تمام کامپوننت ها و اقلام بازی جهت هماهنگی و مدیریت است
//   حتی جهانی (مرحله) بازی هم در انجین قرار می گیرد
//   =====
class Engine {
    World m_world;

    PhysicsSystem m_physics;
    AISystem m_ai;
    HealthSystem m_health;
    RenderSystem m_render;

    vector<
        pair<string,
            function<void(World&, float)>
        > > m_systems; // سرویس لوکیتور نگه دارنده کال بک آپدیت تمام کامپوننت های بازی

public:
    void init() {

        m_health.init(m_world);

        m_systems.push_back({
            "Physics",
            [this](World& world, float dt) {
                m_physics.update(world, dt);
            }
        });
        m_systems.push_back({
            "AI",
            [this](World& world, float dt) {
                m_ai.update(world, dt);
            }
        });
        m_systems.push_back({
            "Health",
            [this](World& world, float dt) {
                m_health.update(world, dt);
            }
        });
        m_systems.push_back({
            "Render",
            [this](World& world, float dt) {
                m_render.update(world, dt);
            }
        });
    };
};

```

```

        m_world.events().on<DeathEvent>(
            [](const DeathEvent& event) {
                cout << "[EVENT] Entity "
                    << event.entity
                    << " died! Cause: "
                    << event.cause
                    << "\n";
            }
        );

        auto player = m_world.create("player");

        player->addComponent<TransformComponent>();
        player->addComponent<VelocityComponent>();
        player->addComponent<HealthComponent>();

        auto enemy = m_world.create("enemy");

        enemy->addComponent<TransformComponent>();
        enemy->addComponent<AIComponent>();
        enemy->addComponent<HealthComponent>();
    }
    // تابعی که باید در هر فریم ران شود، ورودی این تابع در هر فریم زیاد می شود
    void run(int frames = 3) {
        float dt = 0.016f; // معمولا دی تی بر اساس نرخ فریم واقعی تنظیم می شود
        // اینجا برای تکمیل مثال، یک مقدار فرضی تنظیم می شود
        for (int i = 0; i < frames; ++i) { // ایتريت روی فریم های بازی
            cout << "\n--- Frame "
                << i
                << " ---\n";

            for (auto& [name, fn] : m_systems) // ایتريت روی کال بک های سرویس لوکیتور
                fn(m_world, dt); // منجر به آپدیت همه سیستم های بازی مثل فیزیک و ... می شود
        }
    }
};

int main() { // Composite Root
    Engine engine; // شی موتور برای مدیریت تمام اقسام بازی

    engine.init();

    engine.run(1); // ران کردم فریم اول بازی
    engine.run(2); // ران کردم فریم دوم بازی

    return 0;
}

```

✱ پیاده سازی Data-Oriented ECS (حرفه ای و Cache-Friendly)

این نسخه، گامی به سمت معرفی رویکرد Archetype در معماری ECS است! در این نسخه، به جای استفاده از پوینترها و Map، از آرایه های پیوسته (Struct of Arrays) استفاده می کنیم که سریع ترین حالت ممکن برای CPU است.

```

#include <iostream>
#include <vector>
#include <bitset>
#include <cstdint>
#include <iomanip>
using namespace std;

```

```

// =====
// DATA-ORIENTED ECS (DOD) - SoA (Struct of Arrays)
// =====
enum ComponentID { TRANSFORM = 0,
                   VELOCITY,    //1
                   HEALTH,      //2
                   MAX_COMPONENTS }; //3
// سیگنیچر یا امضا به تعداد کامپوننت های تعریف شده بالا یعنی 3 کامپوننت بالا
// بیت تخصیص می دهد، در ساختار بالا 3 بیت پشت سر هم که سمت راست ترینش معادل ترنسفورم
// و بیت وسط معادل ولوسیتی و سمت چپ ترین معادل سیستم هلث است، سپس آرایه ای از این
// ساختار به تعداد انتیتی ها را در نظر بگیرید که هر خانه مختص یک انتیتی است و با
// مراجعه به سیگنیچر هر خانه، می توان فهمید که انتیتی با آی دی برابر ایندکس آن خانه
// کدام کامپوننت های بالا را دارد (بیت معادلش 1 است) و کدامها را ندارد (بیت معادل 0)
using Signature = bitset<MAX_COMPONENTS>; //bit: 2 1 0

H V T
// استفاده از constexpr الزامی برای bitset و template ها
constexpr size_t MAX_ENTITIES = 10000;
// استفاده از vector به جای array برای جلوگیری از Stack Overflow
struct TransformData {
    vector<float> x;
    vector<float> y;
    vector<float> z;
    TransformData() :
        x(MAX_ENTITIES, 0.0f), y(MAX_ENTITIES, 0.0f), z(MAX_ENTITIES, 0.0f) {}
};

struct VelocityData {
    vector<float> vx;
    vector<float> vy;
    vector<float> vz;
    VelocityData() : vx(MAX_ENTITIES, 0.0f), vy(MAX_ENTITIES, 0.0f), vz(MAX_ENTITIES,
0.0f) {}
};

struct HealthData {
    vector<float> hp;
    vector<float> maxHp;
    HealthData() : hp(MAX_ENTITIES, 0.0f), maxHp(MAX_ENTITIES, 0.0f) {}
};

class DODWorld {
    // استفاده از uint8_t به جای bool چون vector<bool> کند و bit-packed است
    vector<uint8_t> m_active;
    vector<Signature> m_signatures; // تخصیص بیتها به تعداد کامپوننت ها
    // اینجا هر بیت نشانه یک کامپوننت خاص است و 0 و 1 بودن هر بیت یعنی داشتن
    // یا نداشتن کامپوننت معادل در انتیتی خاص
    uint32_t m_entityCount = 0; // آی دی جنیتور از این پروپرتی استفاده می کند

    TransformData m_transforms;
    VelocityData m_velocities;
    HealthData m_health;
public:
    DODWorld() : m_active(MAX_ENTITIES, 0), m_signatures(MAX_ENTITIES) {}

    uint32_t createEntity() {
        if (m_entityCount >= MAX_ENTITIES) return UINT32_MAX;
        uint32_t id = m_entityCount++; // تخصیص آی دی و سپس به واحد افزودن به آن برای آیدی بعدی
        m_active[id] = 1; // اکتیو کردن انتیتی کنونی
        return id;
    }
}

```

```

// ست کردن ترنسفورم برای یک آی دی
void addTransform(uint32_t id, float x, float y, float z) {
    if (id >= MAX_ENTITIES) return;
    m_transforms.x[id] = x;
    m_transforms.y[id] = y;
    m_transforms.z[id] = z;
    m_signatures[id].set(TRANSFORM); // بیت معادل کامپوننت ترنسفورم را برابر 1 می کند
}

// تنظیم کامپوننت ولوسیتی برای انتیتی با آی دی ورودی
void addVelocity(uint32_t id, float vx, float vy, float vz) {
    if (id >= MAX_ENTITIES) return;
    m_velocities.vx[id] = vx;
    m_velocities.vy[id] = vy;
    m_velocities.vz[id] = vz;
    m_signatures[id].set(VELOCITY); // بیت ولوسیتی برابر 1 برای سیگنیچر انتیتی
}

// تنظیم کامپوننت هلث
void addHealth(uint32_t id, float hp, float maxHp) {
    if (id >= MAX_ENTITIES) return;
    m_health.hp[id] = hp;
    m_health.maxHp[id] = maxHp;
    m_signatures[id].set(HEALTH);
}

// حلقه پردازش فیزیک - به شدت Cache-Friendly و آماده برای SIMD
void physicsUpdate(float dt) {
    Signature required; // فیزیک روی اشیای دارای ترنسفرم و ولوسیتی اعمال می شود
    required.set(TRANSFORM); // بیت ترنسفرم سیگنیچر را 1 می کنیم
    required.set(VELOCITY); // بیت مختص ولوسیتی را 1 می کنیم

    for (uint32_t i = 0; i < m_entityCount; i++) {
        if (!m_active[i]) continue; // برای انتیتی های غیر فعال اعمال نمی شود
        if ((m_signatures[i] & required) != required) continue;
        // خط بالا یعنی برای انتیتی هایی که ترنسفورم و ولوسیتی ندارند، فیزیک را اعمال نکن
        // گرانش زمین
        m_velocities.vy[i] -= 9.8f * dt;
        // آپدیت موقعیت
        m_transforms.x[i] += m_velocities.vx[i] * dt;
        m_transforms.y[i] += m_velocities.vy[i] * dt;
        m_transforms.z[i] += m_velocities.vz[i] * dt;
        // برخورد با زمین
        if (m_transforms.y[i] < 0) {
            m_transforms.y[i] = 0;
            m_velocities.vy[i] = 0;
        }
    }
}

// پرینت انتیتی با آی دی ورودی
void printEntity(uint32_t id) {
    if (id >= m_entityCount || !m_active[id]) return; // آی دی نا معتبر
    cout << "Entity " << id << " Pos: ("
        << fixed << setprecision(2) // فاصله گذاری مناسب لاگ
        << m_transforms.x[id] << ", "
        << m_transforms.y[id] << ", "
        << m_transforms.z[id] << ")\n";
}

// پرینت تعداد انتیتی های اکتیو صحنه
void printStats() {
    cout << "\n[DOD] Total entities: " << m_entityCount << "\n";
    int active = 0;
    for (uint32_t i = 0; i < m_entityCount; i++)
        if (m_active[i]) active++;
}

```

```

        cout << "[DOD] Active entities: " << active << "\n";
    }
};

int main() {
    cout << "=== Data-Oriented ECS (DOD) Demo ===\n\n";

    DODWorld world;
    float dt = 0.016f; // ~60 FPS
    // ساخت Entity ها
    uint32_t player = world.createEntity(); // ساخت انتیتی پلیر و دسترسی به آی دی او
    world.addTransform(player, 0.0f, 10.0f, 0.0f); // تنظیم موقعیت مکانی پلیر
    world.addVelocity(player, 2.0f, 5.0f, 0.0f); // تنظیم سرعت در جهات پلیر

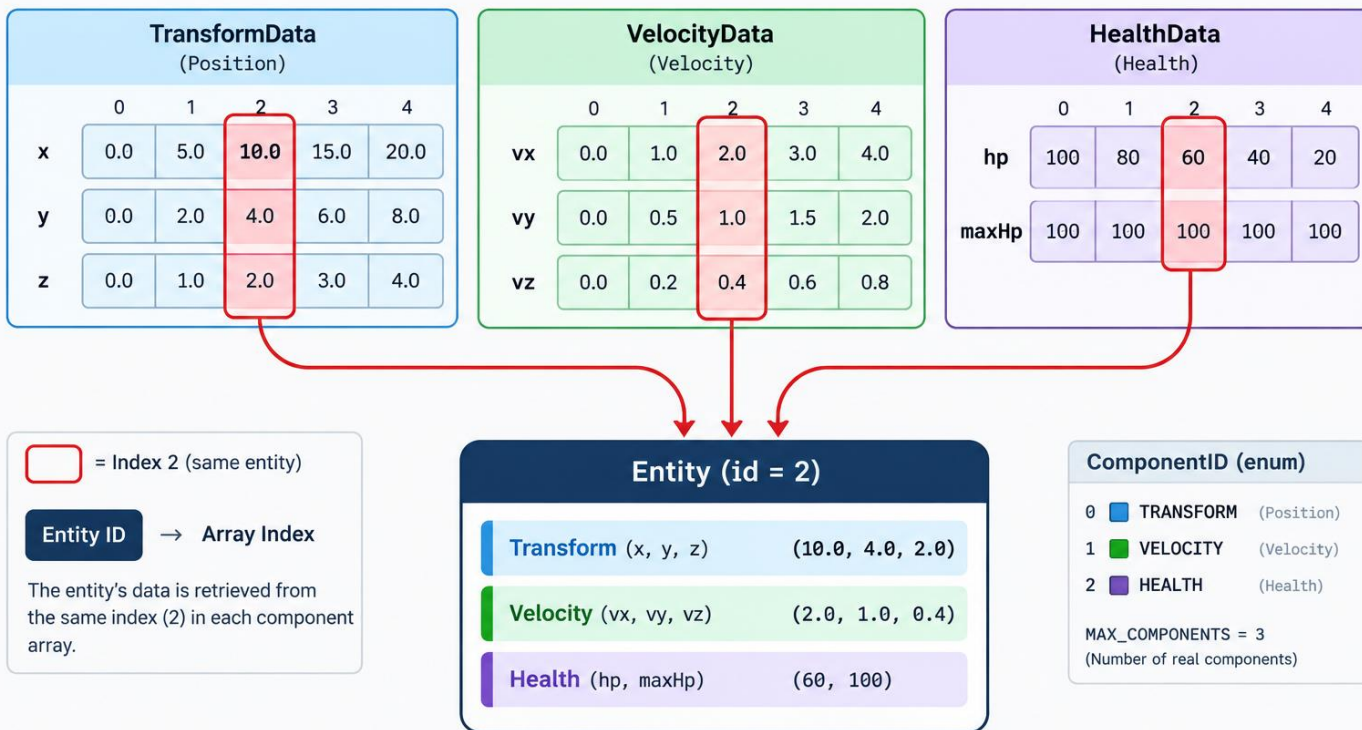
    uint32_t enemy = world.createEntity(); // ساخت انتیتی دشمن
    world.addTransform(enemy, 5.0f, 20.0f, 5.0f); // تنظیم موقعیت مکانی دشمن
    world.addVelocity(enemy, -1.0f, 0.0f, 2.0f); // تنظیم سرعت در جهات دشمن

    world.printStats();
    // شبیه سازی ۳ فریم برای دیدن تاثیر گرانش
    for (int i = 0; i < 3; ++i) {
        cout << "\n--- Frame " << i << " ---\n";
        world.physicsUpdate(dt);
        world.printEntity(player);
        world.printEntity(enemy);
    }
    return 0;
}

```

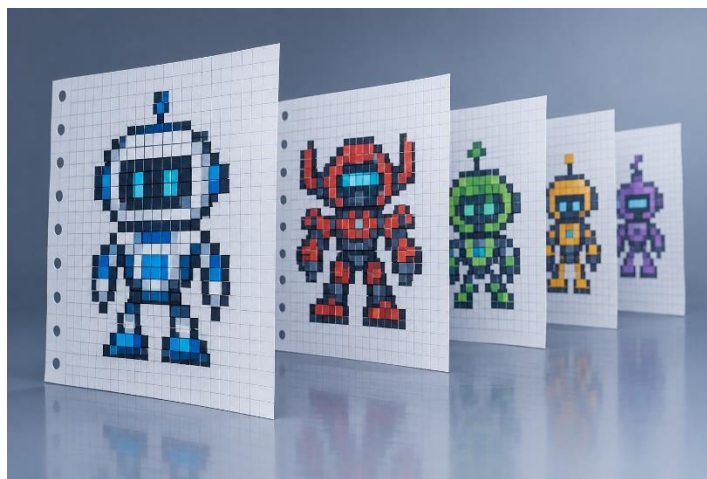
SoA (Structure of Arrays)

Each component type has its own array.
The same index across arrays represents one entity.



توضیحات این نوع پیاده سازی: همچنان که دیدیم، ما با جدا کردن هر Component به یک Struct ویژه آن Component دسترسی

داشتیم و با داشتن آرایه ای از Property های این کامپوننت ها، مشخصات هر Entity را در یک ردیف از آرایه این کامپوننت ها قرار دادیم و با در دست داشتن id هر Entity که در اصل index آن در آرایه هاست، توانستیم با سرعت و پرفرمنس بالا به مشخصات هر کامپوننت دسترسی پیدا کنیم! یعنی مثلاً x یک Entity با id=1 را از ایندکس 1 آرایه x ها در کامپوننت Transform به دست آوردیم و y آن Entity را از ایندکس 1 آرایه y های کامپوننت Transform به دست آوردیم و ... انگار که در تصویر زیرین، ما بخش های مختلف تمام Entity ها را در آرایه های مختلف قرار داده ایم و با در دست داشتن Id هر Entity و رجوع به آرایه های مختلف و رفتن به Index برابر Id هر Entity به مشخصات مختلف آن Entity دسترسی پیدا کنیم و سپس به شکل پازل، Property های هر Entity را از روی این Property ها در کنار هم قرار دهیم و شی مدنظر را بسازیم! ضمناً ما با اشیای Active سر و کار داریم! همچنین آرایه ای از نوع داده بیتی با 3 بیت داده داریم که هر کدام از این بیت ها نماد یکی از 3 کامپوننت مثال ما است و با 1 و 0 شدن هر بیت، متوجه می شویم که آیا ایندکس مدنظرمان دارای کامپوننت هم ارز با آن بیت است یا خیر!



🌱 پیاده سازی Archetype: در پیاده سازی DOD قبلی، داده های هر Component به صورت جداگانه و در قالب آرایه هایی از داده (SoA)

نگهداری شدند. این ساختار باعث بهبود Locality و Cache Friendliness می شود؛ اما در پیاده سازی فعلی، آرایه های Component از ابتدا بر اساس ظرفیت MAX_ENTITIES فضا می گیرند. بنابراین اگر یک Entity، برای مثال فاقد Component ای مانند Transform باشد، همچنان خانه ای متناظر با آن Entity در آرایه ی Transform وجود دارد، اما از آن استفاده نمی شود. این موضوع می تواند باعث مصرف فضای حافظه برای داده هایی شود که برای بعضی Entity ها مورد نیاز نیستند.

علاوه بر این، سیستم برای پیدا کردن Entity های مناسب برای هر System باید Signature هر Entity را بررسی کند. برای مثال، System فیزیک باید بررسی کند که آیا هر Entity دارای Transform و Velocity است یا خیر.

در معماری Archetype، این ایده یک گام جلوتر می رود! Entity هایی که مجموعه Component های یکسانی دارند، در یک Archetype مشترک قرار می گیرند. در نتیجه، Archetype فقط داده Component هایی را نگهداری می کند که واقعاً برای Entity های موجود در آن مورد نیاز هستند. برای مثال، Entity ای که فقط Transform و Render دارد، در Archetype ای قرار می گیرد که فقط همین Component ها را نگهداری می کند و دیگر نیازی به اختصاص فضای Velocity یا Health برای آن Entity وجود ندارد.

از طرف دیگر، چون Entity های دارای مجموعه Component های یکسان از قبل در یک گروه قرار گرفته اند، System ها دیگر مجبور نیستند در هر فریم Signature تک تک Entity ها را بررسی کنند! System فیزیک می تواند مستقیماً Archetype هایی را پردازش کند که دارای Transform و Velocity هستند.

بنابراین می‌توان گفت Archetype ECS اصول Data-Oriented Design و SoA را یک گام جلوتر می‌برد: نه تنها داده‌ها را بر اساس نوع Component سازمان دهی می‌کند، بلکه Entityهای دارای ساختار داده‌ای مشابه را نیز در گروه‌های مجزا قرار می‌دهد. نتیجه، کاهش داده‌های بلااستفاده، بهبود Locality و کاهش هزینه‌ی بررسی Entityهای نامرتبط در Systemها است.

```
#include <iostream>
#include <vector>
#include <unordered_map>
#include <bitset>
#include <cstdint>
#include <memory>
#include <string>
#include <iomanip>

using namespace std;
// =====
//          ARCHETYPE ECS - DOD BASED
// =====
// در DOD قبلی
// Entity -> Signature
// Archetype در
// Signature -> Archetype -> Entities
// Entityهایی که دقیقاً Componentهای یکسان دارند، داخل یک
// Archetype قرار می‌گیرند.
// هر Archetype نیز داده‌های Componentها را به صورت SoA
// نگهداری می‌کند
// =====
// Component IDs
// =====
enum ComponentID {
    TRANSFORM = 0,
    VELOCITY, // 1
    HEALTH, // 2
    RENDER, // 3
    AI, // 4

    MAX_COMPONENTS // 5
};
// =====
// Signature
// =====
using Signature = bitset<MAX_COMPONENTS>; // بیت نماینده یک کامپوننت است
// =====
// Entity
// =====
using Entity = uint32_t;
constexpr Entity INVALID_ENTITY = UINT32_MAX; // کران بالای اعداد بدون علامت صحیح مثبت 32بیتی
// =====
// Components
// =====
struct Transform {
    float x = 0.0f;
    float y = 0.0f;
    float z = 0.0f;
};
struct Velocity {
    float vx = 0.0f;
    float vy = 0.0f;
    float vz = 0.0f;
};
```

```

struct Health {
    float hp = 100.0f;
    float maxHp = 100.0f;
};

struct Render {
    string mesh;
};

struct AI {
    int state = 0;
};

// =====
// Archetype
// =====
// هر Archetype یک Signature مشخص دارد
// مثال
// Signature:
// Transform + Velocity + Health + Render
// Archetype:
// [Transform][Velocity][Health][Render]
// Entity 0
// Entity 1
// Entity 2
// داده ها به شکل SoA نگهداری می شوند
// Transform:
// x[] y[] z[]
// Velocity:
// vx[] vy[] vz[]
// Health:
// hp[] maxHp[]
// Render:
// mesh[]
// =====

class Archetype {
private:
    Signature signature_;
    // -----
    // Entity storage
    // -----
    vector<Entity> entities_; // آرایه ای از uint32_t
    // -----
    // SoA Component Storage
    // -----
    vector<Transform> transforms_;
    vector<Velocity> velocities_;
    vector<Health> healths_;
    vector<Render> renders_;
    vector<AI> ais_;
public:
    explicit Archetype(const Signature& signature) : signature_(signature){}
    // =====
    // Signature
    // =====
    const Signature& signature() const {
        return signature_;
    }
}

```

```

// =====
// Entity count
// =====
size_t size() const {
    return entities_.size();
}
// =====
// Add Entity
// =====
size_t addEntity(
    Entity entity,

    const Transform* transform = nullptr,
    const Velocity* velocity = nullptr,
    const Health* health = nullptr,
    const Render* render = nullptr,
    const AI* ai = nullptr
) {
    size_t index = entities_.size();
    entities_.push_back(entity);
    // اینجا بود و نبود کامپوننت ها در وهله دوم مهم است، اول باید ببینیم بر اساس
    // سیگنیچر آیا اصلا کامپوننت مد نظر در این انتیتی مجاز است و سپس بررسی بود و نبود
    // -----
    // Transform
    // -----
    if (signature_.test(TRANSFORM)) {
        transforms_.push_back(
            transform ? *transform : Transform{}
        );
    }
    // -----
    // Velocity
    // -----
    if (signature_.test(VELOCITY)) {
        velocities_.push_back(
            velocity ? *velocity : Velocity{}
        );
    }
    // -----
    // Health
    // -----
    if (signature_.test(HEALTH)) {
        healths_.push_back(
            health ? *health : Health{}
        );
    }
    // -----
    // Render
    // -----
    if (signature_.test(RENDER)) {
        renders_.push_back(
            render ? *render : Render{}
        );
    }
}

```

```

// -----
// AI
// -----
if (signature_.test(AI)) {
    ais_.push_back(
        ai ? *ai : AI{}
    );
}
return index;
}
// =====
// Accessors
// =====
Transform& transform(size_t index) {
    return transforms_[index];
}
Velocity& velocity(size_t index) {
    return velocities_[index];
}
Health& health(size_t index) {
    return healths_[index];
}
Render& render(size_t index) {
    return renders_[index];
}
AI& ai(size_t index) {
    return ais_[index];
}
Entity entity(size_t index) const {
    return entities_[index];
}
// =====
// Physics System
// =====
void physicsUpdate(float dt) {
    if (!signature_.test(TRANSFORM) ||
        !signature_.test(VELOCITY)) {
        return;
    }

    for (size_t i = 0; i < entities_.size(); ++i) {
        velocities_[i].vy -= 9.8f * dt;

        transforms_[i].x += velocities_[i].vx * dt;

        transforms_[i].y += velocities_[i].vy * dt;

        transforms_[i].z += velocities_[i].vz * dt;
        // Ground collision
        if (transforms_[i].y < 0.0f) {
            transforms_[i].y = 0.0f;
            velocities_[i].vy = 0.0f;
        }
    }
}
// =====
// Health System
// =====
void healthUpdate() {
    if (!signature_.test(HEALTH))
        return;
}

```

```

        for (size_t i = 0; i < entities_.size(); ++i) {
            if (healths_[i].hp <= 0.0f) {
                cout << "Entity "
                    << entities_[i]
                    << " is DEAD\n";
            }
        }
    }

    // =====
    // Render System
    // =====
    void renderUpdate() {
        if (!signature_.test(TRANSFORM) ||
            !signature_.test(RENDER)) {
            return;
        }

        for (size_t i = 0; i < entities_.size(); ++i) {
            cout<< "Entity "
                << entities_[i]
                << " ["
                << renders_[i].mesh
                << "] Position: ("
                << fixed
                << setprecision(2)
                << transforms_[i].x
                << ", "
                << transforms_[i].y
                << ", "
                << transforms_[i].z
                << ")\n";
        }
    }

    // =====
    // Print Archetype Information
    // =====
    void printInfo() const {
        cout << "Archetype Signature: "
            << signature_
            << " | Entities: "
            << entities_.size()
            << '\n';
    }
};

// =====
// World
// =====
//World مسئول
// 1. ساخت Entity
// 2. پیدا کردن Archetype مناسب
// 3. ایجاد Archetype در صورت نیاز
// 4. اجرای System ها
// =====
class World {
private:
    Entity nextEntity_ = 0;
    // -----
    //تمام Archetype های World
    // -----
    vector<unique_ptr<Archetype>> archetypes_;

```

```

// -----
//Archetype به Signature نگاشت
//string به صورت Signature ، برای سادگی آموزشی،
// به عنوان کلید استفاده شده است.
// -----
unordered_map<string, Archetype*> archetypeMap_;
// -----
// Entity -> Archetype
// مشخص میکند هر Entity در کدام Archetype قرار دارد
// -----
unordered_map<Entity, Archetype*> entityArchetype_;
// -----
// Entity -> index inside Archetype
// -----
unordered_map<Entity, size_t> entityIndex_;
private:
    string signatureKey(const Signature& signature) const {
        return signature.to_string();
    }
public:
    // =====
    // Create Entity
    // =====
    Entity createEntity() {
        return nextEntity++;
    }
    // =====
    // Get or Create Archetype
    // =====
    Archetype& getOrCreateArchetype( const Signature& signature) {
        string key = signatureKey(signature);
        auto it = archetypeMap_.find(key);
        if (it != archetypeMap_.end()) {
            return *it->second;
        }
        // جدید Archetype
        archetypes_.push_back(
            make_unique<Archetype>(signature)
        );
        Archetype* archetype = archetypes_.back().get();
        archetypeMap_[key] = archetype;
        return *archetype;
    }
    // =====
    // Add Entity
    // =====
    void addEntity(
        Entity entity,
        const Signature& signature,

        const Transform* transform = nullptr,
        const Velocity* velocity = nullptr,
        const Health* health = nullptr,
        const Render* render = nullptr,
        const AI* ai = nullptr
    ) {
        Archetype& archetype = getOrCreateArchetype(signature);

```

```

        size_t index = archetype.addEntity(
            entity,
            transform,
            velocity,
            health,
            render,
            ai
        );

        entityArchetype_[entity] = &archetype;

        entityIndex_[entity] = index;
    }
    // =====
    // Systems
    // =====
    void physicsUpdate(float dt) {
        for (auto& archetype : archetypes_) {
            archetype->physicsUpdate(dt);
        }
    }
    void healthUpdate() {
        for (auto& archetype : archetypes_) {
            archetype->healthUpdate();
        }
    }
    void renderUpdate() {
        for (auto& archetype : archetypes_) {
            archetype->renderUpdate();
        }
    }
    // =====
    // Debug
    // =====
    void printArchetypes() const {
        cout
            << "\n==== Archetypes =====\n";
        for (const auto& archetype : archetypes_) {
            archetype->printInfo();
        }
    }
};
// =====
// MAIN
// =====
int main() {
    Cout << "=== Archetype ECS Demo ===\n\n";
    World world;
    // =====
    // Archetype A
    // Transform + Velocity + Health + Render
    // Player
    // Enemy1
    // =====
    Signature characterSignature;
    characterSignature.set(TRANSFORM);
    characterSignature.set(VELOCITY);
    characterSignature.set(HEALTH);
    characterSignature.set(RENDER);

```



```

// -----
// Player
// -----
Entity player = world.createEntity();

Transform playerTransform{
    0.0f,
    10.0f,
    0.0f
};

Velocity playerVelocity{
    2.0f,
    5.0f,
    0.0f
};

Health playerHealth{
    100.0f,
    100.0f
};

Render playerRender{
    "Player"
};

world.addEntity(
    player,
    characterSignature,
    &playerTransform,
    &playerVelocity,
    &playerHealth,
    &playerRender
);

// -----
// Enemy1
// -----
Entity enemy1 = world.createEntity();

Transform enemy1Transform{
    5.0f,
    20.0f,
    5.0f
};

Velocity enemy1Velocity{
    -1.0f,
    0.0f,
    2.0f
};

Health enemy1Health{
    80.0f,
    80.0f
};

Render enemy1Render{
    "Enemy1"
};

```

```

world.addEntity(
    enemy1,
    characterSignature,
    &enemy1Transform,
    &enemy1Velocity,
    &enemy1Health,
    &enemy1Render
);
// =====
// Archetype B
// Transform + Render
// Rock
// =====
Signature rockSignature;
rockSignature.set (TRANSFORM);
rockSignature.set (RENDER);

Entity rock = world.createEntity();

Transform rockTransform{
    20.0f,
    0.0f,
    10.0f
};
Render rockRender{
    "Rock"
};

world.addEntity(
    rock,
    rockSignature,
    &rockTransform,
    nullptr,
    nullptr,
    &rockRender
);
// =====
// Archetype C
// Transform + Velocity + Health + Render + AI
// Enemy2
// =====
Signature enemyAISignature;

enemyAISignature.set (TRANSFORM);
enemyAISignature.set (VELOCITY);
enemyAISignature.set (HEALTH);
enemyAISignature.set (RENDER);
enemyAISignature.set (AI);

Entity enemy2 = world.createEntity();

Transform enemy2Transform{
    50.0f,
    15.0f,
    0.0f
};

```

```

Velocity enemy2Velocity{
    -2.0f,
    0.0f,
    0.0f
};

Health enemy2Health{
    120.0f,
    120.0f
};

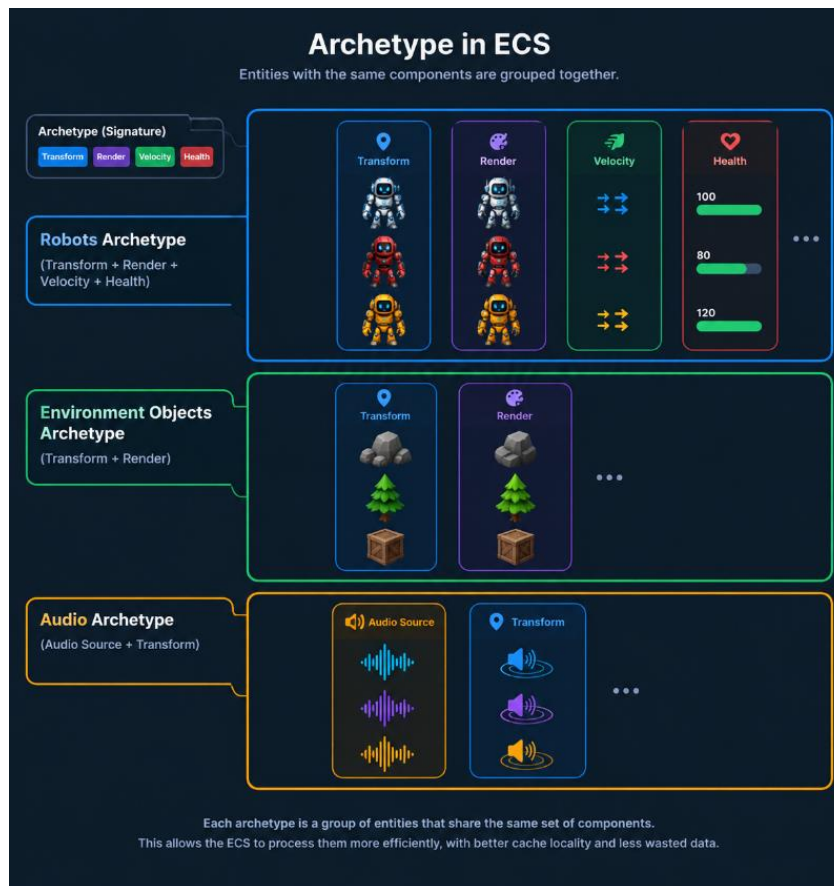
Render enemy2Render{
    "Enemy2"
};

AI enemy2AI{
    1
};

world.addEntity(
    enemy2,
    enemyAISignature,
    &enemy2Transform,
    &enemy2Velocity,
    &enemy2Health,
    &enemy2Render,
    &enemy2AI
);
// =====
// Show Archetypes
// =====
world.printArchetypes();
// =====
// Game Loop
// =====
constexpr float dt = 0.016f;
for (int frame = 0; frame < 3; ++frame) {
    cout << "\n==== Frame "
        << frame
        << "====\n";
    // Physics
    world.physicsUpdate(dt);
    // Health
    world.healthUpdate();
    // Render
    world.renderUpdate();
}
return 0;
}

```

همچنان که شاهد بودید، در رویکرد `Archetype`، ما `unordered_map`ی با کلید `Signature` و `value` نوع `Archetype` داریم! هر `Archetype` شامل آرایه‌ای از تمام `Component`هاست و بر اساس `Signature` خود، به شرط ورود کامپوننت مجاز که `not-null` باشد، آنرا در آرایه `Component` از نوع کامپوننت ورودی اضافه خواهد کرد!



پس از آن، هر Archetype شامل آرایه هایی از کامپوننت های مجازش است و اشیاء در عرض آرایه ها نهفته اند و با استفاده از آرایه entity می توان به آی دی هر Entity بر اساس Index شی در آرایه دسترسی پیدا کرد! پس هر Archetype یک مجموعه از اشیاء بازیست که تعداد و نوع Component هایشان مثل هم ولی با مقادیر مختص خود است! مثل دسته رباتهای هوشمند و دسته اشیاء دارای فیزیک و دسته اصوات قابل تنظیم مکانی در صحنه (مثل صدای جیرجیرک در مکانی خاص در جنگل) و ...

✓ مزیت بسیار مهم Archetype و قسمت بسیار مهم System :

فرض کنیم PhysicsSystem فقط Entity هایی را می خواهد که Transform + Velocity دارند. به جای اینکه تمام Entity های دنیا را بررسی کنیم، می توانیم مستقیماً Archetype مناسب را پردازش کنیم.

```
void physicsUpdate(World& world, float dt)
{
    for (auto& archetype : world.getArchetypes())
    {
        if (!archetype->hasTransform ||
            !archetype->hasVelocity)
            continue;

        for (size_t i = 0;
             i < archetype->size();
             ++i)
        {
            auto& transform = archetype->transforms[i];
            auto& velocity = archetype->velocities[i];
            transform.x += velocity.vx * dt;
        }
    }
}
```

```

transform.y += velocity.vy * dt;

transform.z += velocity.vz * dt;
}
}
}

```

یکی از مشکلات Archetype اینست که اگر در وسط بازی، آرایش کامپوننت های یک Entity تغییر یافت، نمی توانیم صرفاً کامپوننت جدید را به آن اضافه یا از آن کم کنیم! بلکه باید از Archetype قبلی به Archetype جدید move کنیم و این اندکی هزینه پردازشی دارد!

🌱 **پیاده سازی Sparse Set (سریع ترین روش اضافه/حذف):** این الگو (محبوب در کتابخانه EnTT) برای اضافه و حذف کامپوننت در زمان اجرا (Runtime) با پیچیدگی $O(1)$ و تکرار (Iteration) بسیار سریع طراحی شده است.

🌀 **مدیریت اشیای بازی با استفاده از آرایه های Dense (متراکم) و Sparse (پراکنده):** فرض کنید در یک بازی وارد یک مرحله جدید شده اید! کل Entity های داخل این مرحله از ابتدا تا انتها مثلاً 100 عدد Entity با آی دی های بین 0 تا 99 است! البته لزومی ندارد همگی آنها در یک آن واحد قابل رویت باشند! پس از شروع مرحله، کمی در صحنه بازی به جلو حرکت می کنید تا اینکه تعدادی از دشمنان در یک لحظه وارد صحنه می شوند! فرض کنید در این لحظه تعداد Entity های Active بازی اعم از Entity شخصیت شما و دشمنان مجموعاً 20 عدد Entity فعال هستند و بقیه فعلاً Active نیستند و یا قبلاً Active بوده اند و دیگر نیستند و یا در ادامه بازی Active خواهند شد! اینجا ما دو آرایه داریم!

1- **آرایه Sparse (پراکنده):** با نوع داده int با سائیزی برابر تعداد کل Entity های ممکن که همان عدد 100 (فرضی) می باشد! در این آرایه هر Entity در خانه با index مساوی با ID اش قابل رصد است! چگونه؟ Value این آرایه، همان Index همان Entity در آرایه بعدی یعنی Dense (متراکم) است! پس برای Get یا Set کردن مقادیر روی کامپوننت های هر Entity، به سادگی می توان با سرعت $O(1)$ فعال بودن یا نبودنش همراه با Index آنها در آرایه Dense به دست آورد! پس اگر Value یک Index در آرایه Sparse برابر 1- باشد یعنی این شی در حال حاضر Active نیست و اگر هم عددی بزرگتر یا مساوی 0 باشد، یعنی در حال حاضر در index برابر با آن عدد در آرایه Dense قرار دارد که می توان با مراجعه به همان Index در آرایه Dense به مقادیر آن دست پیدا کرد!

2- **آرایه Dense (متراکم):** با نوع داده Entity که تمام اشیای Entity فعال (Active) در صحنه در این آرایه قرار می گیرند! هر شی Entity جدیدی که ساخته می شود، پس از گرفتن ID اش، در اولین خانه خالی آرایه Dense قرار می گیرد و شماره این خانه خالی که اینک متعلق به شی جدید است، در آرایه Sparse و در ایندکس برابر ID این شی جدید قرار می گیرد! ولی در زمان از بین بردن این شی، آخرین شی واقع در آرایه Dense را با این شی جایگزین می کنیم و در این زمان، دو خانه از آرایه Sparse را ویرایش می کنیم! یکی خانه جدید شی منتقل شده را ویرایش می کنیم و index مکان جدید آن در Dense را در خانه با index مساوی با ID آن در Sparse را قرار می دهیم! همچنین Index برابر با ID شی نابود شده در آرایه Sparse را برابر با 1- می کنیم تا معنی InActive بودن این Entity را معنی دهد!

🎮 **مثال:** در آرایه Dense 50 تایی معمولی اگر بخواهید Entity شماره 10 را از وسط آرایه حذف کنید، برای حفظ پیوستگی آرایه Dense باید 40 عنصر بعدی را یکی یکی یک خانه به عقب بکشید (Shift کنید) و این کار زمان بر است!

✅ **در Sparse-Dense:** برای حذف، فقط آخرین عنصر آرایه Dense را برمی داریم و جای عنصر حذف شده می گذاریم (Swap). سپس سائز آرایه را یکی کم می کنیم. زمان حذف و اضافه می شود: $O(1)$.

اکنون سیستم «حرکت» (Movement System) می خواهد فقط position و velocity را آپدیت کند.

وقتی CPU می‌خواهد position یک Entity را بخواند، در ساختار OOP چون داده‌ها در یک struct بزرگ (مثلاً 200 بایتی) پشت سر هم هستند، CPU مجبور است کل آن 200 بایت را از رم به Cache بیاورد، در حالی که فقط به 24 بایت (دو تا Vector3) نیاز دارد! نتیجه: 90٪ داده‌هایی که وارد کش CPU می‌شوند، دور ریخته می‌شوند (Cache Miss) و اتلاف شدید پهنای باند حافظه).

حالت دوم: آرایه 50 تایی در ECS (نگاه DOD با Sparse-Dense)

در ECS، ما داده‌ها را بر اساس نوع کامپوننت جدا می‌کنیم. آرایه Dense شما برای سیستم حرکت، فقط و فقط شامل Position و Velocity است:

```
#include <iostream>
#include <vector>
#include <functional>
#include <cstdint> // uint32_t و UINT32_MAX برای
using namespace std;
// =====
// SPARSE SET - O(1) Insert/Remove, Cache-Friendly Iteration
// =====
template<typename T>
class SparseSet {
    vector<uint32_t> m_sparse; // index = EntityID, value = dense index
    vector<uint32_t> m_dense; // لیست EntityID های فعال
    vector<T> m_data; // موازی با m_dense داده‌ها
    // استفاده از constexpr به جای const
    static constexpr uint32_t INVALID = UINT32_MAX;
public:
    void insert(uint32_t entityId, T data) {
        if (entityId >= m_sparse.size()) {
            m_sparse.resize(entityId + 1, INVALID);
        }
        if (m_sparse[entityId] != INVALID) return; // قبلاً اضافه شده

        m_sparse[entityId] = static_cast<uint32_t>(m_dense.size());
        m_dense.push_back(entityId);
        m_data.push_back(std::move(data));
    }

    void remove(uint32_t entityId) {
        if (entityId >= m_sparse.size() || m_sparse[entityId] == INVALID) return;

        uint32_t denseIdx = m_sparse[entityId];
        uint32_t lastEntity = m_dense.back();
        // Swap with last element for O(1) removal
        m_dense[denseIdx] = lastEntity;
        m_data[denseIdx] = std::move(m_data.back());
        m_sparse[lastEntity] = denseIdx;

        m_dense.pop_back();
        m_data.pop_back();
        m_sparse[entityId] = INVALID;
    }

    T* get(uint32_t entityId) {
        if (entityId >= m_sparse.size() || m_sparse[entityId] == INVALID) return nullptr;
        return &m_data[m_sparse[entityId]];
    }

    bool has(uint32_t entityId) const {
        return entityId < m_sparse.size() && m_sparse[entityId] != INVALID;
    }
};
```

```

size_t size() const {
    return m_dense.size();
}

void forEach(std::function<void(uint32_t, T&)> fn) {
    for (size_t i = 0; i < m_dense.size(); i++) {
        fn(m_dense[i], m_data[i]);
    }
}

};

int main() {
    cout << "=== SparseSet ECS Demo ===\n\n";

    struct Position { float x, y, z; };
    SparseSet<Position> positions;
    //O(1) اضافه کردن
    positions.insert(0, {10.0f, 20.0f, 30.0f});
    positions.insert(5, {50.0f, 60.0f, 70.0f});
    positions.insert(3, {15.0f, 25.0f, 35.0f});

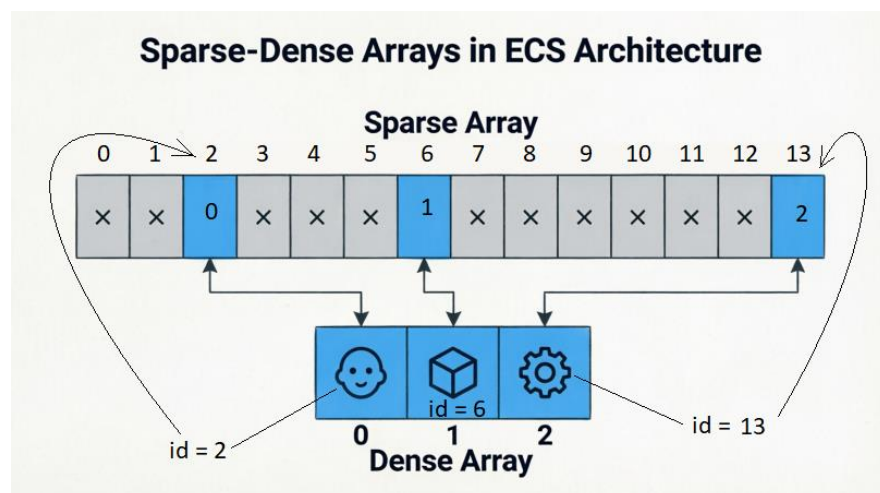
    cout << "Size after inserts: " << positions.size() << "\n";

    cout << "\n--- Iterating (Cache-Friendly) ---\n";
    positions.forEach([](uint32_t id, Position& p) {
        cout << "Entity " << id << ": (" << p.x << ", " << p.y << ", " << p.z <<
    "\n";
    });
    //حذف O(1) با تکنیک Swap-and-Pop
    cout << "\n--- Removing Entity 5 ---\n";
    positions.remove(5);

    cout << "Size after removal: " << positions.size() << "\n";
    cout << "Has Entity 5? " << (positions.has(5) ? "Yes" : "No") << "\n";

    cout << "\n--- Iterating again ---\n";
    positions.forEach([](uint32_t id, Position& p) {
        cout << "Entity " << id << ": (" << p.x << ", " << p.y << ", " << p.z <<
    "\n";
    });
    return 0;
}

```



World: در تمام ساختارهای موجود ECS شاهد بودیم کلاس World مدیریت اصلی اشیای بازی را در دست داشت و می توان آنرا در داخل کلاس Engine قرار داد تا آنرا مدیریت کند! مثال زیر را در نوشتن World برای مدیریت روش Archetype مشاهده می فرمایید!

```
class World {
private:
    std::vector<std::unique_ptr<Archetype>> archetypes;

public:
    Archetype* createArchetype(
        bool transform,
        bool velocity,
        bool health,
        bool render,
        bool ai
    ) {
        archetypes.push_back(
            std::make_unique<Archetype>(
                transform,
                velocity,
                health,
                render,
                ai
            )
        );
        return archetypes.back().get();
    }

    const auto& getArchetypes() const {
        return archetypes;
    }
};
```

ECS در موتورهای واقعی صنعت بازی: امروزه ECS در قلب بزرگترین موتورهای بازی سازی جهان می تپد: 

1- Unity DOTS (Data-Oriented Technology Stack): یک پیاده سازی کامل از Archetype ECS که برای بازی های استراتژیک و شبیه سازی های ترافیک با مقیاس بالا (Massive Scale) طراحی شده است.

2- Unreal Engine 5(Mass Entity): موتور بزرگ آنریل به صورت پیش فرض OOP است، اما برای سیستم های Crowds and Traffic (جمعیت و ترافیک) از فریم ورک ECS به نام Mass Entity استفاده می کند تا هزاران عابر پیاده را همزمان شبیه سازی کند.

3- EnTT(C++): محبوب ترین کتابخانه ECS مستقل در C++ که از الگوی Sparse Set استفاده می کند و به دلیل سرعت سرسام آور و API زیبا، در بسیاری از استودیوهای آینده و تجاری استفاده می شود.

4- Bevy Engine (Rust): یک موتور بازی سازی مدرن که از پایه و اساس بر روی ECS ساخته شده است.

- 1- کامپوننت‌ها را کوچک نگه دارید و به جای یک کامپوننت بزرگ `CharacterStats`، آن را به کامپوننت‌های مختلف همچون `Health`, `Mana`, `Stamina` تقسیم کنید تا `System`‌ها فقط داده‌های مورد نیاز را در کش بارگذاری کنند.
- 2- از پوینترها در کامپوننت پرهیز کنید! کامپوننت‌ها باید `Trivially Copyable` باشند. اگر نیاز به ارجاع به `Entity` دیگری دارید، به جای پوینتر، `Entity ID` آن را ذخیره کنید! این مورد در بخش دسترسی به `Aggregate`‌ها در بخش `DDD` و الگوهای وابسته به `DDD` نیز مشابه این قسمت (استفاده از `ID` به جای `Reference`) بود!
- 3- سیستم‌ها باید `Stateless` باشند! تا حد امکان، `System`‌ها نباید متغیرهای داخلی (`Member Variables`) داشته باشند. تمام وضعیت باید در کامپوننت‌ها ذخیره شود تا موازی‌سازی (`Multithreading`) تضمین شود.
- 4- از `Tag Components` استفاده کنید! گاهی یک `Entity` فقط نیاز به یک "برچسب" دارد. مثلاً: `struct IsDead {};` این کامپوننت داده‌ای ندارد (`Zero-sized`) اما به `System`‌ها اجازه می‌دهد با `Query` کردن آن، موجودیت مرده را شناسایی کنند!

❌ چه زمانی از ECS استفاده نکنیم؟

- پروژه کوچک است (کمتر از 100 Object).
- آبجکت‌ها خیلی `Unique` هستند و رفتار مشترک ندارند!
- تیم کوچک است و `Deadline` نزدیک است! (منحنی یادگیری `ECS` تند است!)
- برای پیاده‌سازی `UI` و منوها (`ECS` برای `UI` یک کابوس است؛ معمولاً در موتورهای `UI` با `MVC` یا الگوهای دیگر مدیریت می‌شود).
- به طور کلی، پیشنهاد می‌شود بازی‌هایتان را با موتورهایی مانند `Unreal Engine` یا `Unity3D` بسازید تا درگیر بسیاری از پیچیدگی‌های زیرساختی نشوید. این موتورهای طی سال‌ها توسعه، هزاران مسئله و چالش ریز و درشت را حل کرده‌اند؛ از `Replication` در بازی‌های `Multiplayer` گرفته تا رندرینگ پیشرفته و بسیاری از امکانات دیگر.
- بنابراین، زمانی که یک بازی را با استفاده از یک `Game Engine` توسعه می‌دهید، به جای اینکه بخش زیادی از زمان خود را صرف طراحی معماری و پیاده‌سازی زیرساخت‌ها و نیازهای اولیه کنید، می‌توانید تمرکز خود را روی طراحی و پیاده‌سازی سناریو و منطق اصلی بازی در سطوح بالاتر قرار دهید.
- دهه‌هاست که تیم‌های مختلف روی چالش‌های گوناگون توسعه بازی کار کرده‌اند و این موتورها را به‌مرور گسترش داده‌اند. هزاران تیم بازی‌سازی با این موتورها کار کرده‌اند و در مواجهه با مشکلات و چالش‌های مختلف، راه‌حل‌های متعددی را آزموده و بهبود داده‌اند. در طول این سال‌ها، روش‌های مختلف بارها پیاده‌سازی، ارزیابی و اصلاح شده‌اند تا این موتورها به شکل امروزی خود برسند.
- بنابراین، اگر هدف شما تولید یک بازی با کیفیت بالا، مشکلات کمتر و سرعت توسعه بیشتر است، استفاده از موتورهایی مانند `Unreal Engine` و `Unity3D` می‌تواند بسیاری از پیچیدگی‌های زیرساختی را از دوش شما بردارد و به شما اجازه دهد بیشتر روی خود بازی تمرکز کنید.



ECS در موتورهای واقعی: 🗝️

Unreal Mass Entity 🌱

```
#include "MassEntityPublicTypes.h"
#include "MassProcessor.h"
// Unreal Engine 5 Mass Entity
USTRUCT()
struct FTransformFragment : public FMassFragment {
    GENERATED_BODY()
    FTransform Transform;
};

UCLASS()
class UMovementProcessor : public UMassProcessor {
    GENERATED_BODY()

    FMassEntityQuery MovementQuery;

    virtual void Execute(
        FMassEntityManager& EntityManager,
        FMassExecutionContext& Context) override {
        EntityManager.ForEachEntityChunk(
            MovementQuery, Context,
            [](FMassExecutionContext& Ctx) {
                // پردازش chunk ها
            }
        );
    }
};
```

```

using Unity.Entities;
using Unity.Mathematics;

// Unity ECS
public struct Translation : IComponentData {
    public float3 Value;
}

public struct Velocity : IComponentData {
    public float3 Value;
}

public class MovementSystem : SystemBase {
    protected override void OnUpdate() {
        float dt = Time.DeltaTime;
        Entities
            .WithAll<Translation, Velocity>()
            .ForEach((ref Translation t, in Velocity v) => {
                t.Value += v.Value * dt;
            })
            .ScheduleParallel(); //! اجرای موازی خودکار!
    }
}

```

EnTT به عنوان محبوب ترین C++ ECS Library 

```

#include <entt/entt.hpp>
#include <iostream>

using namespace std;
// تعریف Component ها
struct TransformComponent { float x, y, z; };
struct VelocityComponent { float vx, vy, vz; };

int main() {
    // ساخت registry
    entt::registry registry;
    // ساخت entity
    auto entity = registry.create();
    // اضافه کردن component
    registry.emplace<TransformComponent>(entity, 0.f, 0.f, 0.f);
    registry.emplace<VelocityComponent>(entity, 1.f, 0.f, 0.f);

    float dt = 0.016f;
    // iterate خیلی سریع
    auto view = registry.view<TransformComponent, VelocityComponent>();
    view.each([dt](auto& t, auto& v) {
        t.x += v.vx * dt;
        t.y += v.vy * dt;
    });
    // group سریعتر از view
    auto group =
        registry.group<TransformComponent>(entt::get<VelocityComponent>);

    return 0;
}

```

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;
// =====
// مشکل (۱) Diamond Problem: مشکل الماس در ارث‌بری چندگانه ✕
// =====
// کلاس پایه
class GameObject {
public:
    virtual void update(float dt) {
        cout << "[GameObject] update()\n";
    }
};
// کلاس‌های میانی که از GameObject ارث می‌گیرند
class Bullet : public GameObject {
public:
    void update(float dt) override {
        cout << "[Bullet] حرکت تیر با سرعت بالا\n";
    }
};
class Light : public GameObject {
public:
    void update(float dt) override {
        cout << "[Light] سوسو زدن نور\n";
    }
};
class SoundEmitter : public GameObject {
public:
    void update(float dt) override {
        cout << "[SoundEmitter] پخش صدای شلیک\n";
    }
};
// =====
// کلاس LightBullet از ۳ کلاس ارث می‌برد ✕
// که همه آن‌ها از GameObject ارث برده‌اند!
// این Diamond Problem است: ۳ نسخه از update() داریم!
class LightBullet : public Bullet, public Light, public SoundEmitter {
public:
    // اگر این override را ننویسیم، کامپایلر خطا می‌دهد
    // "request for member 'update' is ambiguous"
    void update(float dt) override {
        // مجبوریم دستی مشخص کنیم کدام update را می‌خواهیم!
        Bullet::update(dt);
        Light::update(dt);
        SoundEmitter::update(dt);
    }
};
// =====
// مشکل (۲) Bloated Base Class: کلاس پایه متورم ✕
// =====
// فرض می‌کنیم این کلاس‌ها قبلاً تعریف شده‌اند
struct Vector3 { float x, y, z; };
struct Mesh { string name; };
```

```

class GameObjectBloated {
public:
    // با گذشت زمان همه چیز اینجا جمع می شود!
    Vector3 position;
    Vector3 rotation;
    Mesh* mesh;
    float health;
    float damage;
    bool isPhysicsEnabled;
    bool isRendered;
    bool hasAI;
    bool isNetworked;
    bool isSoundEmitter;
    bool isInventory;
    bool isQuestGiver;
    bool isInteractable;
    // 500 خط دیگر ...
    virtual void update() {}
    virtual void render() {}
    virtual void onHit() {}
    virtual void onDeath() {}
    virtual void onPickup() {}
    virtual void onUse() {}
    // 200 متد دیگر ...
};

// Rock به سنگ است! نه health دارد نه AI نه damage
// ولی مجبور است همه این حافظه را carry کند!
class Rock : public GameObjectBloated {
    // فیلدها خالی و اتلاف حافظه!
};

class Enemy : public GameObjectBloated {
    // Enemy به همه اینها نیاز دارد
};

// =====
// راه حل (ECS) فقط چیزی که نیاز داریم ✓
// =====
// کامپوننت‌های کوچک و مستقل
struct TransformComponent { Vector3 pos; Vector3 rot; };
struct MeshComponent { string meshName; };
struct HealthComponent { float hp; float maxHp; };
struct AIComponent { string behavior; };
struct SoundComponent { string soundFile; };
struct LightComponent { float intensity; };
// Entity فقط یک ID است (بدون داده اضافی!)
using EntityID = uint32_t;
// =====
// MAIN - مقایسه عملی
// =====
int main() {
    cout << "\n";
    cout << " OOP vs ECS - OOP Problems \n";
    cout << "\n\n";
    // — Demonstrating Problem 1: Diamond Problem —
    cout << "=== Problem 1: Diamond Problem ===\n";
    LightBullet lb;
    lb.update(0.016f);

    cout << "\n👉 We had to manually specify which update() should be called!\n";
    cout << " In large projects, this becomes a maintenance nightmare.\n\n";
}

```

```
// — Demonstrating Problem 2: Bloated Base Class —
cout << "=== Problem 2: Bloated Base Class ===\n";
cout << "sizeof(GameObjectBloat) = " << sizeof(GameObjectBloat) << " bytes\n";
cout << "sizeof(Rock) = " << sizeof(Rock) << " bytes\n";
cout << "sizeof(Enemy) = " << sizeof(Enemy) << " bytes\n\n";

cout << "👉 A simple rock (Rock) occupies exactly the same memory\n";
cout << "   as an intelligent enemy (Enemy)!\n";
cout << "   With 1000 rocks in the game = " << 1000 * sizeof(Rock)
    << " bytes of wasted memory!\n\n";

// — Demonstrating the Solution: ECS —
cout << "=== Solution: ECS ===\n";
cout << "sizeof(EntityID) = " << sizeof(EntityID) << " bytes (just an ID)\n";
cout << "sizeof(TransformComponent) = " << sizeof(TransformComponent) << " bytes\n";
cout << "sizeof(MeshComponent) = " << sizeof(MeshComponent) << " bytes\n";
cout << "sizeof(HealthComponent) = " << sizeof(HealthComponent) << " bytes\n";
cout << "sizeof(AIComponent) = " << sizeof(AIComponent) << " bytes\n";

cout << "👉 In ECS:\n";
cout << "   • A rock only has: EntityID + Transform + Mesh\n";
cout << "   • An enemy only has: EntityID + Transform + Mesh + Health + AI\n";
cout << "   • Each Entity has exactly what it needs!\n";
cout << "   • No wasted memory, no Diamond Problem, no bloated classes!\n";

return 0;
}
```

خروجی این کد: 

OOP vs ECS – OOP Problems

=== Problem 1: Diamond Problem ===

[Bullet] حرکت تیر با سرعت بالا
[Light] سوسو زدن نور
[SoundEmitter] پخش صدای شلیک

👉 We had to manually specify which update() should be called!
In large projects, this becomes a maintenance nightmare.

=== Problem 2: Bloated Base Class ===

sizeof(GameObjectBloat) = 56 bytes
sizeof(Rock) = 56 bytes
sizeof(Enemy) = 56 bytes

👉 A simple rock (Rock) occupies exactly the same memory
as an intelligent enemy (Enemy)!
With 1000 rocks in the game = 56000 bytes of wasted memory!

=== Solution: ECS ===

sizeof(EntityID) = 4 bytes (just an ID)
sizeof(TransformComponent) = 24 bytes
sizeof(MeshComponent) = 32 bytes
sizeof(HealthComponent) = 8 bytes
sizeof(AIComponent) = 32 bytes

👉 In ECS:

- A rock only has: EntityID + Transform + Mesh
- An enemy only has: EntityID + Transform + Mesh + Health + AI
- Each Entity has exactly what it needs!

مقایسه ECS با سایر الگوها:

معیار	 OOP + Hierarchy	 Component Pattern	 ECS مدرن
انعطاف پذیری	کم 	متوسط 	زیاد 
Cache Friendliness	بد 	متوسط 	عالی 
Testability	سخت 	متوسط 	آسان 
Parallelism	سخت 	سخت 	آسان 
پیچیدگی	کم 	متوسط 	متوسط 
Debugging	آسان 	متوسط 	سخت تر 
مناسب برای	پروژه‌های کوچک	پروژه‌های میانی	پروژه‌های بزرگ

نتیجه: 

✓ سرعت بالا (Cache Friendly)

✓ انعطاف زیاد (امکان داشتن ترکیبات گوناگون با حذف Diamond Problem)

✓ تست آسان (هر System مستقل است)

✓ موازی‌سازی آسان (System ها مستقلن)

✓ حذف Bloated Base Class

درک ECS، درک نحوه کار CPU در سطح پایین، و پذیرش طراحی داده‌محور (DOD)، مرز بین یک "برنامه‌نویس بازی" و یک "مهندس موتور بازی (Engine Programmer)" را مشخص می‌کند!

 **جمع‌بندی:** معماری ECS فقط یک ترند زودگذر نیست؛ بلکه تکامل طبیعی معماری نرم‌افزار برای مواجهه با محدودیت‌های سخت‌افزاری مدرن (به‌ویژه شکاف سرعت بین CPU و RAM) است.

1- الگوهای همزمانی و برنامه‌نویسی ناهمگام (Concurrency & Asynchronous Patterns):

منبع اصلی این خانواده کتابی با عنوان Pattern-Oriented Software Architecture, Vol. 2 (POSA2) است.

پترن	تعریف
Active Object	فراخوانی متد را از اجرای واقعی آن جدا می‌کند؛ هر شیء یک Thread اجرای مستقل دارد و درخواست‌ها را در یک صف نگه می‌دارد تا به ترتیب پردازش شوند.
Reactor	یک Event Loop تکی روی یک Thread، رویدادهای I/O چندگانه را دریافت و به Handler مناسب Dispatch می‌کند مدل هسته Node.js و Nginx
Proactor	مشابه Reactor، اما عملیات I/O به صورت Async توسط خود سیستم‌عامل انجام می‌شود و فقط بعد از اتمام کامل عملیات، Callback صدا زده می‌شود مدل Windows IOCP
Balking	اگر شیء در وضعیت نامناسبی برای انجام عملیات باشد، عملیات فوراً و بدون انتظار رد می‌شود.
Guarded Suspension	برخلاف Balking، درخواست به جای رد شدن، تا برقراری شرط لازم Block/Suspend می‌شود.
Thread Pool	استخر ثابتی از Thread های ازپیش ساخته شده که Task ها را اجرا می‌کنند؛ از هزینه ساخت/نابودی مکرر Thread جلوگیری می‌کند. (بیس آن در مورد Object Pool در کتاب بررسی شده است!)
Worker Thread	یک Thread اختصاصی که کار پس‌زمینه را مستقل از Thread اصلی انجام می‌دهد؛ پایه مفهومی Thread Pool
Half-Sync/Half-Async	یک لایه Synchronous برای منطق برنامه و یک لایه Asynchronous برای I/O، با یک صف واسط بین این دو.
Leader/Followers	چند Thread به نوبت نقش «رهبر» (گوش دادن به رویداد بعدی) را برعهده می‌گیرند؛ جایگزین سبک‌تری برای Reactor تک Threaded
Master/Worker	یک Master کار را به قطعات مستقل تقسیم و به چند Worker می‌سپارد، سپس نتایج را جمع‌آوری می‌کند.
Pipeline	پردازش در چند مرحله متوالی (Stage) که هر کدام می‌تواند روی Thread/Process جدا اجرا شود؛ خروجی هر مرحله ورودی مرحله بعد است.
Promise	نماینده یک نتیجه که در آینده آماده می‌شود؛ پایه مفهومی الگوی Async/Await در زبان‌های مدرن.
Event-based Asynchronous Pattern (EAP)	الگوی قدیمی‌تر عمدتاً از دنیای .NET. برای مدل‌سازی عملیات Async از طریق رویداد Completed به جای Callback مستقیم.

2- الگوهای ارتباطی و (Session Communication Patterns)

پترن	تعریف
Session Pattern (Stateful vs Stateless)	تصمیم معماری درباره اینکه آیا سرور باید وضعیت مکالمه را بین درخواست‌های متوالی یک کلاینت نگه دارد (Stateful) یا هر درخواست کاملاً مستقل باشد (Stateless) مثل REST
Client Pull (Request-Response)	کلاینت فعالانه درخواست می‌فرستد و منتظر پاسخ می‌ماند؛ مدل ارتباطی رایج HTTP سنتی.
Server Push (Register-Notify)	کلاینت ابتدا برای دریافت رویدادهای آینده ثبت‌نام می‌کند؛ سرور در لحظه وقوع رویداد، خودش به کلاینت اطلاع می‌دهد مثل WebSocket یا Server-Sent Events

3- الگوهای رفتاری/ساختاری تکمیلی (Behavioral & Structural) فراتر از GoF : این‌ها عمدتاً از POSA ، کتاب Refactoring از Fawler و مقالات ایشان و سنت طراحی جاوا اسکریپت/جاوا می‌آیند.

پترن	تعریف
Plug-in	امکان افزودن یا تغییر رفتار برنامه در زمان اجرا یا استقرار، بدون تغییر کد اصلی؛ معمولاً برای برآوردن نیازهای متفاوت هر محیط. (Fowler – PoEAA)
Module	بسته‌بندی کد مرتبط در یک واحد مستقل با کپسوله‌سازی جزئیات داخلی.
Revealing Module	نسخه‌ای از Module Pattern (رایج در جاوا اسکریپت) که فقط بخش عمومی (API) را آشکار می‌کند و بقیه را در Closure پنهان نگه می‌دارد.
Wrapper	اصطلاح عمومی برای پوشاندن یک شیء API/موجود جهت ساده‌سازی، تطبیق، یا افزودن رفتار چتری بالای Adapter ، Decorator و Facade
Mixin	افزودن رفتار به یک کلاس از طریق ترکیب (نه وراثت خطی)؛ راهی برای اشتراک کد بین کلاس‌های غیرمرتبط.
Marker Interface / Marker	یک Interface کاملاً خالی که صرفاً برای «نشانه‌گذاری» یک نوع در سیستم Type استفاده می‌شود مثل Serializable در جاوا
Twin	شبیه‌سازی وراثت چندگانه در زبان‌هایی که آن را پشتیبانی نمی‌کنند، با دو شیء مرتبط که هر کدام یک والد را نمایندگی می‌کنند.
Fluent Interface	سبک طراحی API با Method Chaining برای خوانایی بیشتر کد فراخوانی رایج در Builder های زنجیره‌ای
Specification	کپسوله‌کردن یک قاعده یا شرط کسب‌وکار به صورت یک شیء مستقل و قابل ترکیب (AND/OR/NOT) برای استفاده در اعتبارسنجی، فیلتر، یا انتخاب.
Servant	رفتار مشترک بین چند کلاس نامرتبط را در یک شیء کمکی بیرونی قرار می‌دهد، بدون افزودن آن رفتار به سلسله‌مراتب کلاس‌ها.
Extension Object	افزودن قابلیت جدید به یک شیء در زمان اجرا، بدون تغییر تعریف کلاس اصلی آن.

پترن	تعریف
Double Dispatch	انتخاب متد نهایی بر اساس نوع Runtime دو شیء درگیر (نه فقط یکی)؛ پایه فنی الگوی Visitor
Blackboard	چند «منبع دانش» (Knowledge Source) «مستقل» از طریق یک فضای داده مشترک (Blackboard) با هم همکاری می کنند؛ برای مسائل پیچیده بدون راه حل الگوریتمی مشخص (مثل تشخیص گفتار).
Event Aggregator	یک واسط مرکزی که چند منبع رویداد را جمع می کند و به مشترکین اطلاع می دهد؛ وابستگی مستقیم بین Publisher ها و Subscriber های متعدد را کاهش می دهد.
Publish-Subscribe	تولیدکننده پیام (Publisher) بدون آگاهی از مشترکین، پیام را منتشر می کند؛ مشترکین (Subscriber) بر اساس موضوع (Topic) آن را دریافت می کنند. پایه اصلی اکثر سیستم های Messaging مدرن Kafka ، RabbitMQ نیز همین الگوست.

4- الگوهای سازمانی (J2EE (Core J2EE Patterns & Fowler): منبع اصلی: کتاب Core J2EE Patterns و بخشی از Patterns of Enterprise Application Architecture مارتین فاولر.

پترن	تعریف
Business Delegate	لایه واسطی بین Presentation و منطق کسب و کار سمت سرور که جزئیات فراخوانی Remote Lookup، Retry، خطا را از کلاینت پنهان می کند.
Data Access Object (DAO)	انتزاع کامل دسترسی به یک منبع داده پشت یک Interface مشخص؛ جد مفهومی الگوی Repository امروزی.
Interceptor	افزودن رفتار جانبی Logging، Security، Transaction قبل یا بعد از اجرای یک عملیات، بدون تغییر کد خود عملیات.
Service Stub	پیاده سازی سبک و جایگزین برای یک سرویس خارجی گران قیمت، کند، یا غیر قابل دسترس — معمولاً برای تست یا محیط توسعه.
Data Transfer Object (DTO)	شیء ساده و بدون منطق برای انتقال داده بین لایه ها یا از طریق شبکه؛ کاهش تعداد Call Remote ها.
Data Transfer Object Assembler	مسئول ساخت یک یا چند DTO از روی مدل دامنه (و گاهی برعکس).
Service Layer	لایه ای که مرز دقیق Use Case های برنامه، منطق تراکنش، و امنیت را برای کلاینت ها تعریف می کند.
Remote Facade	یک Facade درشت دانه (Coarse-Grained) روی مجموعه ای از اشیای ریزدانه Fine-Grained Remote. برای کاهش تعداد فراخوانی های Remote.
Value List Handler	مدیریت اجرای Query های بزرگ و صفحه بندی (Pagination) نتایج بدون بارگذاری کل داده در حافظه.

5- الگوهای ORM و پایداری داده: (Fowler – Patterns of Enterprise Application Architecture)

مرجع اصلی همه این موارد، فصل Data Source Architectural Patterns و Object-Relational کتاب PoEAA فاولر است.

پترن	تعریف
Table Module	یک کلاس به ازای هر جدول پایگاه داده که منطق کسب و کار مربوط به کل جدول (نه یک ردیف خاص) را نگه می دارد.
Table Data Gateway	یک شیء واحد که تمام دسترسی SQL به یک جدول را کپسوله می کند یک Instance برای کل جدول
Row Data Gateway	یک شیء به ازای هر ردیف جدول، با متدهایی برای دسترسی/تغییر همان ردیف مشخص.
Foreign Key Mapping	نگاشت یک رابطه بین دو شیء به یک ستون کلید خارجی در پایگاه داده رابطه ای.
Association Table Mapping	نگاشت یک رابطه چند-به-چند (Many-to-Many) از طریق یک جدول واسط مستقل.
Dependent Mapping	یک شیء وابسته (که هویت مستقلی ندارد) از طریق شیء صاحبش (Owner) ذخیره و بازیابی می شود، بدون Mapper اختصاصی خودش.
Embedded Value	نگاشت یک Value Object مثل Money یا Address به چند ستون از همان جدول شیء صاحب آن.
Serialized LOB	ذخیره یک گراف پیچیده از اشیاء به صورت یک فیلد Serialized شده (XML/JSON/Blob) در یک ستون واحد.
Single Table Inheritance	نگاشت کل سلسله مراتب وراثت به یک جدول واحد، با یک ستون تشخیص نوع (Discriminator).
Class Table Inheritance	یک جدول جداگانه برای هر کلاس در سلسله مراتب وراثت.
Concrete Table Inheritance	یک جدول جداگانه برای هر کلاس Concrete (نه انتزاعی)، شامل تمام فیلدهای به ارث رسیده از والدین.
Inheritance Mappers	ساختاردهی Mapper ها به گونه ای که منطق مشترک نگاشت وراثت (هر کدام از سه الگوی بالا) در یک مکان متمرکز شود.
Layer Supertype	یک کلاس پایه مشترک که همه اشیای یک لایه از آن ارث می برند (مثلاً یک Entity پایه با فیلد Id
Mapper	شیء واسطی که داده را بین دو زیرسیستم مستقل مثلاً Domain و Database جابه جا می کند، بدون اینکه آن دو زیرسیستم یکدیگر را بشناسند.
Data Mapper	نوع خاصی از Mapper که Domain Model را کاملاً مستقل از ساختار پایگاه داده نگه می دارد پایه ORM های مدرن مثل Hibernate/EF Core
Identity Map	برای هر رکورد بارگذاری شده در یک تراکنش، فقط یک نمونه در حافظه نگه می دارد تا از بارگذاری تکراری و ناهمخوانی داده جلوگیری شود.

پترن	تعریف
Lazy Loading	تأخیر در بارگذاری داده مرتبط (مثلاً یک رابطه) تا لحظه‌ای که واقعاً به آن نیاز باشد. (ما در این کتاب در مورد Lazy Initialization به عنوان Base این الگو بحث کرده ایم!)
Query Object	نمایش یک Query پایگاه داده به صورت یک شیء قابل ترکیب و قابل استفاده مجدد، به جای رشته SQL خام. (مطلب مرتبط در پترن Interpreter که در کتاب بررسی شده است!)

6- الگوهای معماری UI و اپلیکیشن (UI / Application Architecture)

پترن	تعریف
Front Controller	یک نقطه ورودی واحد برای مدیریت متمرکز همه درخواست‌های وب Routing، احراز هویت، Logging مشترک
Page Controller	برخلاف Front Controller، یک Controller مستقل برای هر صفحه یا Action وب.
Service Locator	رجیستری مرکزی برای دریافت سرویس‌ها به جای ساخت مستقیم آن‌ها؛ الگویی بحث‌برانگیز که در مقاله جداگانه ما با جزئیات (و نقد مارک سیمن/فاولر) بررسی شد. (در کتاب ما بحث شده)
Separated Interface	تعریف یک Interface در یک پکیج مجزا از پیاده‌سازی‌اش، برای کاهش وابستگی زمان کامپایل بین لایه‌ها.
Gateway	شیء ساده‌ای که دسترسی به یک سیستم یا API خارجی را پشت یک Interface یکنواخت کپسوله می‌کند.

7- الگوهای معماری مدرن و ابری (Cloud-Native / Distributed Systems)

پترن	تعریف
Circuit Breaker	بعد از چند شکست پیاپی در فراخوانی یک سرویس، فراخوانی‌های بعدی موقتاً بدون تلاش واقعی رد می‌شوند تا از فروپاشی آبشاری (Cascading Failure) جلوگیری شود.
Bulkhead	ایزوله کردن منابع (Thread Pool)، (Connection Pool) بین بخش‌های مختلف سیستم، تا شکست یک بخش باعث از کارافتادن کل سیستم نشود؛ در Health Check نیز برای جداسازی بررسی سلامت هر زیرسیستم استفاده می‌شود.
Backend for Frontend (BFF)	یک لایه Backend اختصاصی و سبک برای هر نوع کلاینت (وب، موبایل، ...) که دقیقاً نیازهای همان کلاینت را برآورده می‌کند.
Heartbeat / Health Check	ارسال دوره‌ای سیگنال «زنده‌ام» یا فراهم کردن یک Endpoint بررسی سلامت، برای تشخیص سریع خرابی یک سرویس.

8- الگوهای پیام‌رسانی و قابلیت اطمینان (Messaging & Reliability): Publish-Subscribe که در بخش ۳ تعریف

شد، از نظر کاربرد عملی مستقیماً به این خانواده هم مرتبط است!

پترن	تعریف
Inbox / Outbox Pattern	تضمین تحویل قابل اعتماد پیام هم‌زمان با تغییر پایگاه داده: پیام در یک جدول محلی (Outbox) در همان تراکنش دیتابیس ذخیره می‌شود و یک پردازش جداگانه، بعداً آن را واقعاً ارسال می‌کند؛ از ناسازگاری بین «تغییر دیتابیس» و «ارسال پیام» جلوگیری می‌کند.

9- الگوهای اینترپرایس (Enterprise Integration Patterns (Hohpe & Woolf): کتاب Enterprise

Integration Patterns مرجع اصلی طراحی سیستم‌های پیام‌محور هست و Inbox/Outbox فقط نوک کوه یخ است:

Message Channel, Message Router, Content-Based Router, Splitter/Aggregator, Message Translator, Dead Letter Channel, Competing Consumers, Claim Check, Correlation Identifier

10- الگوهای مایکروسرویس (Chris Richardson / microservices.io): با توجه به اینکه قبلاً درباره

Clean/Onion/Hexagonal/DDD صحبت کردیم، این خانواده منطقاً کنارشان قرار دارد!

Saga, API Gateway, Strangler Fig, Database per Service, Aggregator, Choreography vs Orchestration, Sidecar, Service Mesh

11- Cloud Design Patterns (Microsoft/AWS): پترن‌های تاب‌آوری و مقیاس‌پذیری فراتر از چیزی که در گروه 7

داشتیم:

Anti-, Ambassador, Sharding, Queue-Based Load Leveling, Throttling, Cache-Aside, Retry Federated Identity, Compensating Transaction, Gatekeeper, Corruption Layer

12- Test Doubles (Meszaros - xUnit Test Patterns): در کنار «Service Stub» مورد دیگر از انواع

الگوهای Unit Test وجود دارد:

Mock, Spy, Stub, Dummy, Fake

13- اصطلاحات و Idiom های C++: این‌ها Design Pattern کلاسیک نیستند، بلکه Idiom مخصوص زبان علل الخصوص ++C

هستند:

Copy-and-Swap, Type Erasure, CRTP (Curiously Recurring Template Pattern), RAII

14- الگوهای ضدالگو (Anti-Patterns): در بسیاری از کتابها و مراجع، این الگوها را به عنوان آنتی پترن ها می شناسند:

Anemic Domain Model ،Golden Hammer ،Spaghetti Code ،Big Ball of Mud، God Object

15- الگوهای داده مدرن (تکمیلی بر گروه ۵ و ۸): این گروه، ادامه منطقی گروه الگوهای ORM/Persistence هستند!

الگوی Sharding در این گروه و گروه 10 یکپست!

Command Query Responsibility ،Sharding ،Change Data Capture (CDC) ، Materialized View
Event Sourcing و Segregation

سخنان پایانی نویسنده

از اینکه این کتاب را مطالعه فرمودید بینهایت سپاسگزارم. امیدوارم این کتاب برایتان مفید بوده باشد! گام های احتمالی آینده اینجانب در جهت ارتقای این کتاب و ارائه نسخه های بعدی عبارتند از:

- 1- مقدمه کامل بر چيستی معماری نرم افزار در فصل اول این کتاب
- 2- افزودن مبانی مفید طراحی چارت های UML در فصل دوم این کتاب
- 3- گنجاندن مقاله رویکردهای Solid در فصل سوم
- 4- گنجاندن رویکرد Aspect Oriented Programming و Functional Programming و ...
- 5- افزودن الگوی Saga و الگوی Inbox/OutBox و Publish/Subscriber و برخی الگوهای دیگر ...
- 6- رویکردهای جدیدتر معماری ECS

نوشتن این کتاب مصادف شد با اتمام فصلی از زندگی تخصصی اینجانب در زمینه مهندسی و معماری نرم افزار و ورود به فصل جدیدی از زندگی حرفه ای در زمینه برنامه نویسی هوش مصنوعی!

التماسِ همت برای ساختنِ جهانی بهتر، و دستِ دعایی برای تعجیل در آمدنِ ناجیِ موعود؛

با هر نامی که او را می خوانید، و به هر آیینی که آمدنش را باور دارید.

مباد که تنها چشم به راهِ فردایی روشن باشیم؛ که شایسته تر آن است که خود، آغازگرِ آن فردا باشیم؛

بیایید دست در دستِ هم، از امیدِ پلی به سوی فردا بسازیم؛

فردایی که در آن، عدالت بر ستم، مهر بر کینه، و روشنایی بر تاریکی چیره شود.

و اگر بهشتی در کار باشد، بی گمان خانه ی آنان است که بودندشان، جهان را زیباتر می کند؛

آنان که هر روز، نه تنها خود نیکوتر می شوند، که همچون خورشید بر روشنایی جهان می افزایند،

آزادگانی که از روی عقل و اختیار، اگر هزاران سال مجالِ زیستن یابند،

به جای سو استفاده از عقل و اختیار، هر روز مهربان تر و خردمندتر می شوند و هیچ جانی از حضورشان آزرده نمی شود.

اگر حیات جاویدان بهشتی در کار باشد، سزاوارِ کسانی ست که حضورشان برای جهان:

نه رنج، که رحمت است؛

نه تاریکی، که روشنایی است؛

و نه ویرانی، که خود معنای نجات است.

برای همه شما آرزوی موفقیت در همه شئون زندگی و لیاقت زندگی ابدی را دارم...

التماسِ دعا... ﷻ

هادی عباسی / انتشار در 1 مهر 1405 / September 23rd, 2026