Design patterns

تیم:

ایده پردازان

حوری دهش

رضوان عابدین

ساحل مهدوی نژاد

- آیا تاکنون با وضعیتی روبه‌رو شده‌اید که تغییرات در یک بخش از کد، تغییرات گسترده‌ای را در سایر بخش‌ها نیازمند کرده باشد؟
- آیا تاکنون به کدی برخورد کرده‌اید که نگهداری و تست آن چالش برانگیز بوده است؟
- آیا تاکنون با شرایطی روبه‌رو شده‌اید که توسعه‌پذیری پروژه اهمیت زیادی داشته باشد؟
- آیا تا به حال با مشکلاتی در حوزه مشارکت و کار تیمی در پروژه‌های نرم‌افزاری روبه‌رو شده‌اید؟

دیزاین پترن چیست؟

- راه حل های معمولی برای مشکلات رایج در طراحی نرم افزار
- دیزاین پترن‌ها راه‌حلِ کامل محسوب نمی‌شوند. یعنی شامل کدهای برنامه، کلاس‌ها یا کتابخانه‌هایی نیستند که بتوانیم به طور مستقیم در پروژه‌هایمان استفاده کنیم. بلکه نوعی راهکار برای حل مسئله مشخص به‌شمار می‌روند به عبارت دیگر نمی توان فقط یک الگو را پیدا کرده و در برنامه خود کپی کنیم
  - الگو یک قطعه کد خاص نیست، بلکه یک مفهوم کلی برای حل یک مسئله خاص است
- الگوها اغلب با الگوریتم ها اشتباه گرفته می شوند، زیرا هر دو مفهوم راه حل های معمولی برای برخی از مشکلات شناخته شده را توصیف می کنند:
  - یک الگوریتم همیشه مجموعه ای واضح از اقداماتی را تعریف می کند که می تواند به برخی از اهداف دست یابد
  - یک الگو یک توصیف سطح بالاتر از یک راه حل است
  - کد الگوی یکسانی که برای دو برنامه مختلف اعمال می شود ممکن است متفاوت باشد

الگو از چه چیزی تشکیل شده است؟

- بیشتر الگوها بسیار رسمی توصیف می شوند تا افراد بتوانند آنها را در زمینه های مختلف بازتولید کنند

- بخش هایی که در توضیحات الگو وجود دارند عبارتند از:
  - هدف الگو: به طور خلاصه هم مسئله و هم راه حل را تشریح می کند.
  - انگیزه: بیشتر مسئله را توضیح می دهد و راه حلی را که الگو ممکن می سازد.
  - ساختار کلاس ها: هر بخش از الگو و نحوه ارتباط آنها را نشان می دهد.
  - مثال کد در یکی از زبان های برنامه نویسی محبوب: درک ایده پشت الگو را آسان تر می کند.

چرا باید الگوها را یاد بگیریم؟

- دیزاین پترن ها مجموعه ابزاری از راه حل های آزمایش شده برای مسئله های رایج در طراحی نرم افزار هستند.

- حتی اگر هرگز با این مسئله ها مواجه نشوید، دانستن الگوها همچنان مفید است زیرا به شما می آموزد که چگونه انواع مسئله ها را با استفاده از اصول طراحی شی گرا حل کنید.

- الگوهای طراحی یک زبان مشترک را تعریف می کنند که شما و هم تیمی هایتان می توانید برای برقراری ارتباط موثرتر از آن استفاده کنید.
  - می‌توانید بگویید: «اوه، فقط از یک Singleton برای آن استفاده کنید»
  - همه ایده پشت پیشنهاد شما را درک خواهند کرد.اگر الگو و نام آن را بدانید نیازی به توضیح نیست

دلیل نیاز ما به دیزاین پترن چیست؟

- افزایش قابلیت نگهداری کدهای برنامه
- قابلیت استفاده مجدد
- ترویج بهترین روش‌ها
- تسهیل مشارکت و کار تیمی
- برای به‌کارگیری دیزاین پترن مناسب در پروژه‌هایمان، لازم است تا هدف و کاربرد هر یک از این دیزاین پترن‌ها را درک کرده باشیم.

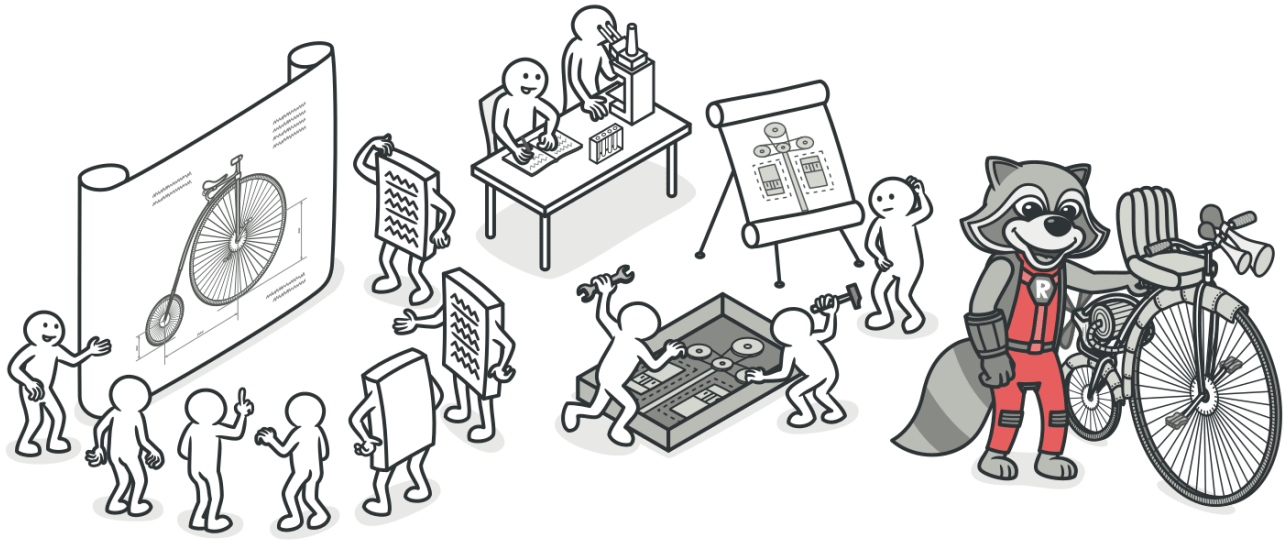
مزایا و معایب استفاده از دیزاین پترن

- **مزایا:**
- سرعت بخشیدن به توسعه
- بهبود کیفیت کدها
- دیباگ آسان‌تر
- ثبات و سازگاری در طراحی

- **معایب:**
- استفاده بی‌رویه یا نابه‌جا
- منحنی یادگیری
- بهینه‌سازی زودرس
- انعطاف پذیری محدود

انواع دیزاین پترن

- همه الگوها را می توان بر اساس هدف یا هدفشان دسته بندی کرد
- دیزاین پترن‌ها را می‌توان به ۳ دسته کلی تقسیم کرد:
  - Creational patterns
  - Structural patterns
  - Behavioral patterns



انواع دیزاین پترن

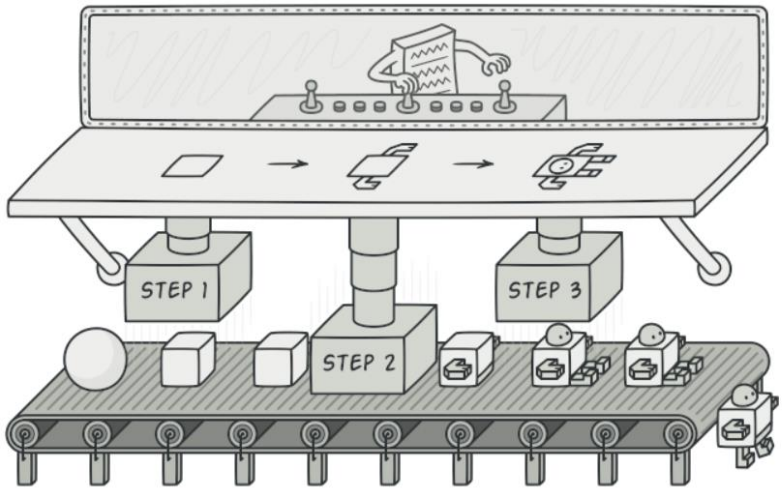
- Creational patterns : این الگوها به ما امکان می‌دهند روند ایجاد و ساخت شیء را مدیریت کرده و از وابستگی‌ها و جزئیات پیچیده آن فاصله بگیریم.

- Structural patterns : این الگوها به ما امکان می‌دهند تا روابط میان اشیاء و کلاس‌ها را بهبود بخشیده و ساختار سیستم را قابل توسعه و قابل استفاده مجدد کنیم.

- Behavioral patterns : این الگوها بر روی رفتار و تعامل بین اجزای سیستم تمرکز دارند و به ما امکان می‌دهند تا تغییرات در رفتار برنامه را به صورت انعطاف‌پذیر اعمال کنیم.

Creational patterns

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton



Builder design pattern

- چرا باید از الگوری طراحی builder استفاده کنیم؟
- راه حل ارائه شده توسط الگوری طراحی builder چیست؟

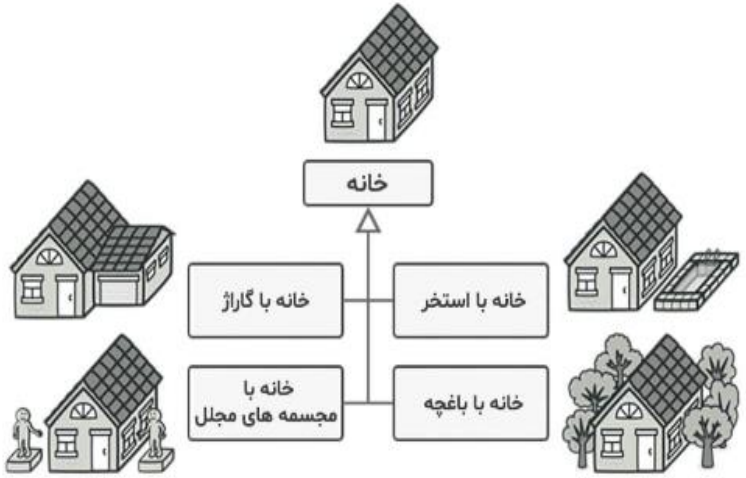
Builder design pattern

چرا باید از الگوری طراحی builder استفاده کنیم؟

گاهی در فرآیند طراحی یک نرم افزار نیاز به ساخت و تعریف اشیایی پیچیده با پارامترهای زیاد خواهیم داشت. عملیات مقداردهی اولیه پارامترهای همچین اشیایی در سازنده ی آن‌ها را تصور کنید. برای این کار باید تعداد زیادی پارامتر را به سازنده آن اشیا ارسال کنیم، که این کار باعث افزایش پیچیدگی و کاهش خوانایی کدها در نرم افزار خواهد شد.

Builder design pattern

به عنوان مثال، فرض کنید که می‌خواهیم یک خانه طراحی و ایجاد کنیم. برای ساخت یک خانه ساده، تنها نیاز به ساخت چهار دیوار و یک کف، نصب درب، جایگذاری پنجره و در نهایت ساخت یک سقف خواهیم داشت. برای این کار باید یک کلاس خانه ایجاد کنیم. اما اگر تصمیم بگیریم که یک خانه بزرگ‌تر و روشن‌تر با یک حیاط خلوت و سایر امکانات رفاهی از جمله سیستم گرمایشی، لوله کشی، سیم کشی برق و... داشته باشیم.



Builder design pattern

آیا به سادگی می‌توانیم همه این امکانات را در کلاس خانه اضافه کنید؟

قطعا خیر! زیرا در آن صورت باید تعداد بسیار زیادی پارامتر را برای مقدار دهی به سازنده کلاس خانه ارجاع دهیم. این عمل باعث افزایش پیچیدگی و کاهش خوانایی کدهای نرم افزار خواهد شد. همچنین امکان دارد در اغلب موارد، بسیاری از پارامترها استفاده نشده باقی بمانند، که باعث کثیف شدن کدنویسی می‌شوند. به عنوان مثال، اگر فقط یک خانه دارای استخر شنا باشد، پارامترهای مربوط به استخر شنا برای ساخت خانه‌های بدون استخر خالی (null) خواهند ماند.

Builder design pattern

راه حل ارائه شده توسط الگوری طراحی builder چیست؟

الگوی سازنده روشی را برای ساخت اشیا پیچیده پیشنهاد می‌دهد که روند ساخت آن‌ها را بسیار ساده‌تر می‌کند. در این روش به جای اینکه پارامترهای زیادی به یک Constructor ارسال شود یا از تعداد زیادی Constructor استفاده شود، کلاسی را ایجاد می‌کنند که وظیفه ساخت اشیا را بر عهده بگیرد. برای این کار باید کدهایی که وظیفه ساخت اشیا را بر عهده دارند از کلاس مربوط به اشیا بیرون کشیده شوند. سپس باید این کدها در کلاسی جدید قرار گیرند که به آن کلاس Builder می‌گویند. از این پس، برای ساخت اشیا از کلاس Builder استفاده می‌شوند.

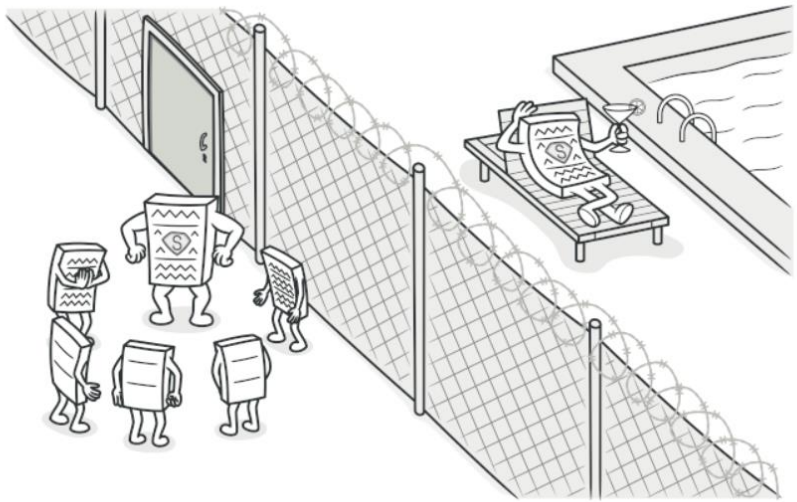
Builder design pattern

راه حل ارائه شده توسط الگوری طراحی builder چیست؟

کلاس builder ساخت بخش‌های مختلف شی را به مجموعه ای از مراحل تقسیم بندی می‌کند. مثلا ساختن دیوارها، درها و... . مزیتی که این روش نسبت به روش‌های قبلی دارد این است که دیگر نیازی به انجام تمام مراحل برای ساخت یک شی نخواهد بود. بر این اساس برای ساخت یک شی با ساختاری خاص، تنها مراحل مورد نیاز انجام می‌شوند. امکان دارد برخی از مراحل ساخت یک شی نسبت به اشیای دیگر متفاوت باشد. به عنوان مثال، دیوارهای یک کلبه ممکن است از چوب ساخته شوند، اما دیوارهای قلعه باید از جنس سنگ باشند.

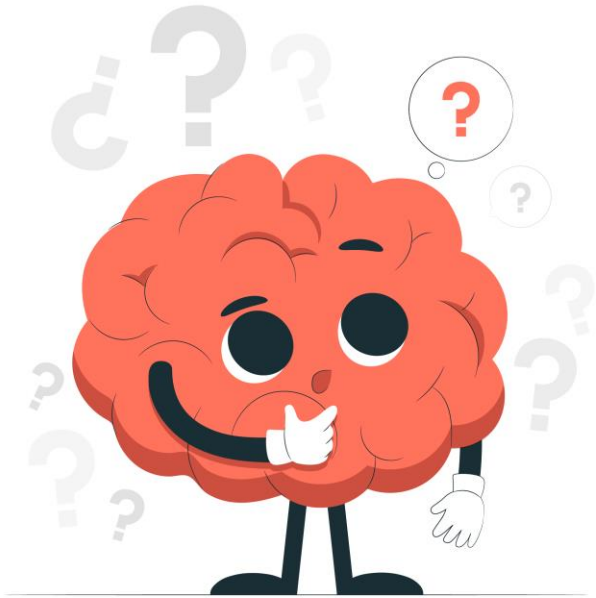
Structural patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy



Proxy design pattern

- چرا باید از الگوری طراحی Proxy استفاده کنیم؟
- چرا میخواهیم دسترسی به یک شی را کنترل کنیم؟
- راه حل ارائه شده توسط الگوری طراحی Proxy چیست؟



Proxy design pattern

چرا باید از الگوری طراحی Proxy استفاده کنیم؟

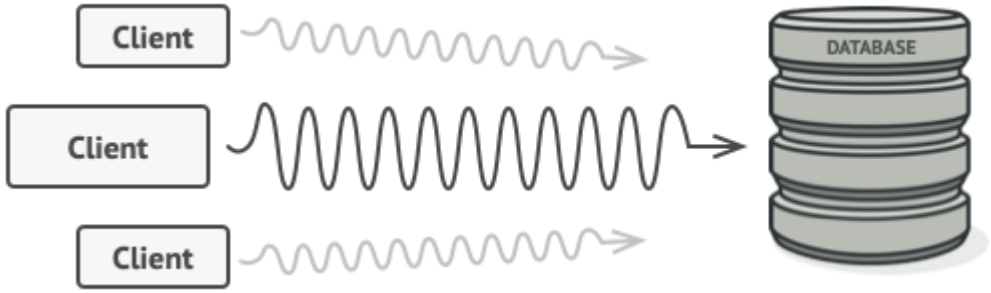
پروکسی یک الگوی طراحی ساختاری است که امکان تهیه یک جایگزین یا مکان نگهدار برای یک شی دیگر را فراهم می‌کند. یک پروکسی دسترسی به شی اصلی را کنترل می‌کند و امکان انجام کارهایی را قبل یا بعد از ارسال درخواست به شی اصلی، می‌دهد.

Proxy design pattern

چرا میخواهیم دسترسی به یک شی را کنترل کنیم؟

فرض کنید، یک شی عظیم داریم که مقدار زیادی از منابع سیستم را مصرف می‌کند و هر از گاهی به آن نیاز داریم، اما نه همیشه.(کوئری های database)

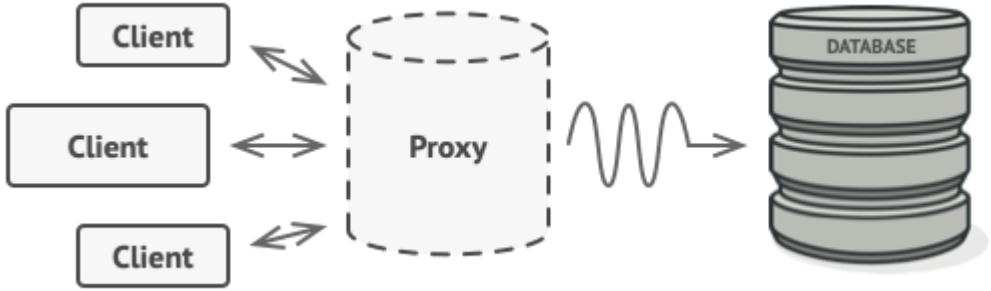
پیاده سازی Lazy initialization: این شی را فقط زمانی ایجاد کنید که واقعاً مورد نیاز است. همه کلاینت های شی باید Lazy initialization معوق را اجرا کنند، که باعث تکرار کدهای زیادی می شود.



Proxy design pattern

راه حل ارائه شده توسط الگوری طراحی Proxy چیست؟

روشی که الگوی پروکسی پیشنهاد می‌کند: یک کلاس پروکسی جدید با رابط مشابه با یک شیء اصلی سرویس ایجاد کنیم، سپس برنامه را به‌روزرسانی می‌کنیم تا شیء پراکسی را به همه کلاینت‌های شی اصلی ارسال کند. با دریافت درخواست از کلاینت، پروکسی یک شیء سرویس واقعی ایجاد می کند و تمام کارها را به آن واگذار می کند.



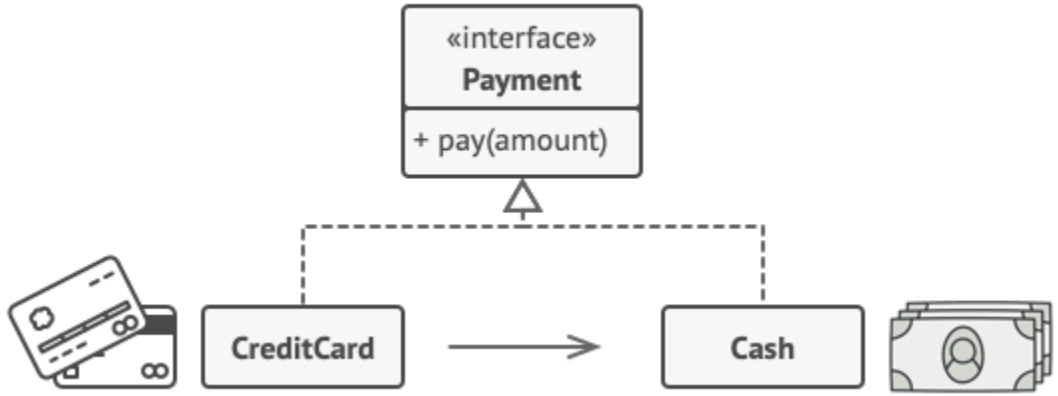
Proxy design pattern

چه مزیتی دارد؟

اگر لازم باشه چیزی را قبل یا بعد از منطق اولیه کلاس اجرا کنیم، با پروکسی میتوانیم این کار را بدون تغییر آن کلاس انجام بدهیم. با توجه به اینکه پروکسی همان رابط کلاس اصلی را پیاده سازی می کند، می توانیم آن را به هر کلاینتی که انتظار یک شیء سرویس واقعی را دارد، ارسال کرد.

Proxy design pattern

مثال برای درک بهتر رابط ها



Behavioral patterns

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

با تشکر از حسن توجه شما