پارت اول

مقدمه

نوشتن کدی که فقط کار کند، کافی نیست. در دنیای واقعی، کدی ارزشمند است که **قابل خواندن، قابل نگهداری، قابل تست و قابل توسعه** باشد. اینجاست که مفهوم **Clean Code** یا کد تمیز وارد می‌شود. در این مستند، قصد داریم اصول کدنویسی تمیز را در زمینه **برنامه‌نویسی شیءگرا (OOP)** بررسی کنیم و در ادامه به اصول **SOLID** بپردازیم که ستون فقرات طراحی خوب در OOP هستند.

فصل ۲: اصول Clean Code در طراحی شیءگرا

۲.۱ اصل خوانایی کد (Code Readability)

خوانایی مهم‌ترین ویژگی یک کد تمیز است. کدی که واضح باشد، راحت‌تر بررسی، دیباگ و توسعه داده می‌شود.

```python
class ReportGenerator:
    def __init__(self, data_source):
        self.data_source = data_source

    def generate_report(self):
        data = self.data_source.get_data()
        formatted_data = self._format_data(data)
        self._export_report(formatted_data)

    def _format_data(self, data):
        # Formatting logic here
        return data

    def _export_report(self, data):
        # Export logic here
        print("Report exported.")
```

توضیح کامل:

- کلاس `ReportGenerator` نامی معنادار دارد که به‌روشنی هدف کلاس را بیان می‌کند. این یکی از اصول بنیادین Clean Code است: نام‌گذاری باید گویای نقش و عملکرد باشد.
- توابع داخلی مثل `_format_data` و `_export_report` با یک آندرلاین شروع شده‌اند که نشان‌دهنده‌ی **سطح دسترسی داخلی** یا protected هستند؛ این یعنی این توابع برای استفاده‌ی داخلی کلاس طراحی شده‌اند.
- متد `generate_report` تمام گام‌های لازم را به‌صورت **ساده و خطی** انجام می‌دهد: گرفتن داده، فرمت کردن، خروجی دادن. این ساختار باعث می‌شود توابع کوچک‌تر و قابل تست‌تر باشند.
- از اصل (Single Responsibility Principle) SRP پیروی شده: هر متد فقط یک کار انجام می‌دهد.
- با تقسیم عملیات به توابع داخلی، اصل Separation of Concerns رعایت شده است. کد در توابع کوچک‌تر تقسیم شده و این باعث افزایش خوانایی و تست‌پذیری می‌شود.
- نام‌گذاری توابع مطابق با کنوانسیون snake_case است. این کنوانسیون در پایتون باعث خوانایی بهتر می‌شود، مخصوصاً در مقایسه با camelCase که در زبان‌هایی مثل جاوا استفاده می‌شود.
- ساختار کلاس به گونه‌ای است که می‌توان در آینده آن را تست یا توسعه داد بدون اینکه نیاز به بازنویسی کلی باشد.
- متغیر `data_source` نشان‌دهنده وابستگی به یک منبع داده است. این وابستگی به‌جای اینکه در داخل کلاس ساخته شود، از بیرون تزریق شده که مفهومی از **Dependency Injection** است و به تست‌پذیری کمک می‌کند.
- توابع داخلی هر کدام فقط یک سطح از abstraction دارند. این یعنی در هر سطح فقط یک لایه از منطق دیده می‌شود، نه جزئیات زیادی که باعث شلوغی شوند.
- کد تمیز یعنی کدی که نیازی به کامنت ندارد. در اینجا به‌دلیل نام‌گذاری درست و تقسیم منطقی توابع، کد تقریباً **self-documenting** است.
- استفاده از underscore در توابع داخلی (مانند `_format_data`) علامتی غیررسمی در پایتون برای internal method است، که به خواننده‌ی کد اطلاع می‌دهد از این متدها خارج از کلاس نباید استفاده کند.
- رعایت فاصله بین توابع و بدنه کلاس به‌گونه‌ای است که خوانایی را افزایش می‌دهد و ساختار کد از هم مجزا و واضح باقی می‌ماند.

- در Clean Code گفته می‌شود که کد باید طوری نوشته شود که انگار قرار است توسط فردی که هیچ‌چیز از پروژه نمی‌داند، خوانده شود. این کلاس نمونه‌ی خوبی از این اصل است.
- همچنین رعایت اصل YAGNI (You Aren't Gonna Need It) مشهود است: هیچ متدی اضافه یا زایدی در این کلاس وجود ندارد و فقط نیازهای واقعی پیاده‌سازی شده‌اند.

۲.۲ اصل کوتاه‌سازی توابع (Small Functions)

توابع باید کوتاه باشند؛ به‌طور کلی توصیه می‌شود هر تابع نهایتاً ۵ تا ۱۵ خط داشته باشد و فقط یک کار انجام دهد. این کار باعث کاهش پیچیدگی و افزایش خوانایی می‌شود.

```
class Invoice:
    def __init__(self, items):
        self.items = items

    def calculate_total(self):
        return sum(self._calculate_item_total(item) for item in self.item

    def _calculate_item_total(self, item):
        return item['price'] * item['quantity']
```

توضیح کامل:

- کلاس `Invoice` یک صورت‌حساب را مدل‌سازی می‌کند. نام آن کوتاه ولی دقیق و حرفه‌ای است.
- متد `calculate_total` وظیفه‌ی محاسبه مجموع کل را دارد و این وظیفه را به متد `_calculate_item_total` تفویض کرده است.
- این تفویض باعث می‌شود هر تابع تنها یک کار انجام دهد (اصل SRP).
- تابع `calculate_total` بسیار کوتاه، ساده و قابل تست است. چنین توابعی کمتر احتمال خطا دارند و به‌راحتی می‌توانند تغییر یا بازطراحی شوند.
- استفاده از generator expression در `sum(...)` باعث شده کد خواناتر، سریع‌تر و Pythonic باشد.
- متد `_calculate_item_total` داخلی است و فقط برای استفاده‌ی داخلی کلاس طراحی شده. علامت underscore اول آن به این موضوع اشاره دارد.

- نام‌گذاری توابع به‌صورت snake_case و متغیرهای معنادار (مانند `item`, `items`, `price`, `quantity`) از اصول مهم Clean Code هستند.
- اگر عملیات محاسبه‌ی قیمت در آینده پیچیده‌تر شود، می‌توان به‌راحتی فقط متد `_calculate_item_total` را تغییر داد، بدون اینکه منطق کلی تابع `calculate_total` تحت تأثیر قرار گیرد.
- اگر تمام عملیات در یک تابع انجام می‌شد، تابع بزرگ‌تر و پیچیده‌تر می‌شد که نگهداری آن سخت‌تر بود.
- توابع کوتاه کمک می‌کنند تا تست‌نویسی ساده‌تر شود، چون می‌توان برای هر بخش جداگانه تست نوشت.
- ساختار کد طوری طراحی شده که قابلیت توسعه داشته باشد. مثلاً می‌توان تخفیف، مالیات یا ... را در توابع جدید اضافه کرد.
- توابع کوچک کمتر نیاز به کامنت دارند، چون وظیفه‌شان واضح است.
- خوانایی کدهای تابع‌محور بسیار بالاست؛ کسی که برای اولین بار این کلاس را می‌بیند به‌راحتی درک می‌کند که چه اتفاقی در حال رخ دادن است.
- همچنین اصل KISS (Keep It Simple, Stupid) در اینجا رعایت شده است: توابع بدون پیچیدگی غیرضروری نوشته شده‌اند.
- در طراحی تمیز توصیه می‌شود توابع مانند مراحل یک دستور آشپزی باشند: قدم به قدم و قابل درک. این ساختار دقیقاً همین حالت را دارد.

پارت دوم

۲.۳ اصل نام‌گذاری معنادار (Meaningful Naming)

نام‌ها باید بازتاب‌دهنده‌ی دقیق هدف متغیر، تابع یا کلاس باشند. نام خوب به تنهایی می‌تواند نیمی از مستندسازی را انجام دهد.

```python
class UserAuthenticator:
    def __init__(self, user_repository):
        self.user_repository = user_repository

    def is_valid_user(self, username, password):
        user = self.user_repository.find_by_username(username)
        return user is not None and user.password == password
```

توضیح کامل:

- نام کلاس `UserAuthenticator` دقیقاً کاری که کلاس انجام می‌دهد را توصیف می‌کند: اعتبارسنجی کاربران.
- استفاده از `is_valid_user` به‌جای `check` یا `validate` باعث شده عملکرد متد کاملاً واضح باشد و حتی در خواندن جملاتی شبیه به انگلیسی هم کمک کند.
- پارامترهای `username` و `password` نیز نام‌هایی کاملاً توصیفی دارند، نه فقط `u` یا `x` که نامفهوم باشند.
- متغیر `user` نیز از لحاظ معنایی صحیح است و نشان می‌دهد که این متغیر چه نوع داده‌ای را در خود دارد.
- نام `user_repository` دقیقاً نشان می‌دهد که این وابستگی قرار است منبعی برای دریافت اطلاعات کاربر باشد، نه صرفاً یک `data` یا `db` مبهم.
- نام‌گذاری متد با پیشوند `is_` نشان‌دهنده‌ی این است که مقدار برگشتی `Boolean` است، که طبق کنوانسیون‌های Python و Clean Code توصیه می‌شود.

- در Clean Code تأکید زیادی روی اجتناب از مخففسازی بی‌مورد وجود دارد. اینجا به‌جای `usrRepo` یا `chkUsr` از نام‌های کامل و واضح استفاده شده است.
- در پایتون توصیه می‌شود که از snake_case برای نام متدها و متغیرها استفاده شود؛ این در این مثال کاملاً رعایت شده.
- نام متد و متغیر باید به توسعه‌دهنده‌ی جدید پروژه کمک کند تا بدون نیاز به دیدن جزئیات عملکرد، بفهمد چه چیزی قرار است انجام شود.
- اگر نام متد `check` بود، توسعه‌دهنده نمی‌دانست قرار است چه چیزی بررسی شود. اما `is_valid_user` کاملاً هدف را مشخص کرده.
- استفاده از `find_by_username` برای متد repository هم پیرو همین اصول است: واضح، دقیق، و در سطح abstraction مناسب.
- اصول Clean Code می‌گویند: اگر نام‌گذاری درست انجام شود، بسیاری از نیازهای کامنت‌نویسی حذف خواهند شد؛ این مثال کاملاً همین را نشان می‌دهد.
- همچنین کد به‌گونه‌ای نوشته شده که حتی بدون کامنت، هر توسعه‌دهنده‌ای بتواند فقط با خواندن نام‌ها عملکرد کلی را بفهمد.
- انتخاب نام خوب یک هنر است و بهبود آن نیاز به تمرین و بازخورد دارد. باید از کلی‌گویی، نام‌های عمومی یا مختصر پرهیز کرد.

۲.۴ اصل اجتناب از تو در تویی بیش از حد (Avoid Deep Nesting)

کدهای تو در تو خوانایی را کاهش می‌دهند و درک و نگهداری آن‌ها دشوار می‌شود. راهکارهایی مثل **early return** می‌توانند ساختار را بهبود بخشند.

```
def process_order(order):
    if order is None:
        return "Invalid order"

    if not order.is_paid:
        return "Order not paid"

    if order.is_cancelled:

```

```
        return "Order was cancelled"

    ship_order(order)
    return "Order shipped"
```

توضیح کامل:

- در نسخه غیرتمیز این کد، ممکن بود چندین `if` تو در تو داشته باشیم که خواندن آن سخت و خسته‌کننده می‌شد.
- استفاده از **early return** یا بازگشت زودهنگام باعث شده ساختار کد **مسطح (flat)** باقی بماند.
- این روش باعث می‌شود توسعه‌دهنده بتواند سریع‌تر منطق را دنبال کند.
- هر شرط فقط همان حالتی را بررسی می‌کند که مانع ادامه روند است و در صورت وقوع، کد بلافاصله متوقف می‌شود.
- تو در تویی بیش از حد معمولاً نشانه‌ی عدم تفکیک وظایف یا استفاده نادرست از شرط‌ها است.
- در Clean Code پیشنهاد می‌شود بلوک‌های شرطی به‌صورت مستقل و بدون تو در تو بودن بررسی شوند، مگر در موارد خاص.
- این مثال به‌جای داشتن ساختار تو در تو شبیه به `if -> if -> if -> else` ، از ساختاری صاف بهره می‌برد که به وضوح کمک می‌کند.
- همچنین، تمام مقادیر برگشتی از جنس `str` هستند که باعث یکنواختی خروجی تابع می‌شود.
- اگر در آینده نیاز باشد شرط دیگری اضافه شود، فقط کافیست یک شرط دیگر در خط جدید نوشته شود، بدون اینکه تو در تویی جدیدی اضافه شود.
- همین ساختار کمک می‌کند خطاها راحت‌تر لاگ شوند یا برای آن‌ها unit test نوشته شود.
- استفاده از نام تابع واضح (`process_order`) و پارامتر شفاف (`order`) هم خوانایی را بالا برده است.
- Clean Code به وضوح تأکید دارد: "کد باید طوری نوشته شود که گویی برای انسان خوانده می‌شود، نه ماشین."
- این تابع دقیقاً همین اصل را رعایت کرده و درک آن حتی برای افراد تازه‌کار نیز راحت است.
- این الگو تقریباً در تمام زبان‌ها قابل استفاده است و یکی از تکنیک‌های ساده اما بسیار مؤثر برای تمیز کردن ساختار شرطی است.

پارت سوم

۲.۵ حذف کد مرده و غیرقابل استفاده (Remove Dead Code)

کدهایی که هرگز اجرا نمی‌شوند یا دیگر کاربردی ندارند، باید حذف شوند. وجود این نوع کدها باعث کاهش خوانایی، سردرگمی توسعه‌دهنده‌ها، و در برخی موارد بروز خطا می‌شود.

```
class Product:
    def __init__(self, name, price):
        self.name = name
        self.price = price
        # self.discount = 0

    def get_final_price(self):
        return self.price

    # def apply_discount(self):
    #     self.price = self.price * 0.9
```

توضیح کامل:

- در این مثال، کدی داریم که یا کامنت شده یا بدون استفاده رها شده است. مثل متغیر `discount` یا متد `apply_discount` .
- اگر نیازی به این ویژگی‌ها نیست، باید کاملاً حذف شوند تا کد تمیز و واضح باقی بماند.
- نگه داشتن کدهای بلااستفاده باعث می‌شود افراد جدید تیم به اشتباه تصور کنند که این بخش‌ها هنوز بخشی از منطق پروژه هستند.
- وجود `apply_discount` به‌صورت کامنت‌شده شاید در ذهن توسعه‌دهنده‌ی اصلی معنی‌دار بوده، اما برای دیگران فقط ابهام ایجاد می‌کند.
- در Clean Code اصل بر این است: "اگر کدی استفاده نمی‌شود، نباید آن را نگه داشت؛ چون هر خط کد یک مسئولیت است."
- متغیر `discount` حتی اگر قرار باشد بعداً استفاده شود، بهتر است در زمان نیاز اضافه گردد، نه به امید استفاده‌ی آینده.

- یکی دیگر از دلایل مهم حذف کدهای مرده، جلوگیری از بروز **conflict** در mergeها و تغییرات آینده است.
- کد مرده می‌تواند باعث شکست تست‌ها، تداخل با refactor آینده، یا کاهش عملکرد تحلیل‌گرهای استاتیک شود.
- اگر بخواهید پروژه را تست‌نویسی کنید، متدهای بدون استفاده باعث پوشش پوشالی می‌شوند و درصد پوشش را بی‌معنا می‌کنند.
- همچنین ممکن است تیم‌های دیگر از روی وجود چنین کدهایی تحلیل نادرستی از روند پروژه برداشت کنند.
- در محیط‌هایی که اتوماسیون و CI/CD فعال است، این کدها می‌توانند مانعی جدی برای pipeline باشند.
- حذف آن‌ها باعث کاهش حجم کد، سرعت بیشتر بررسی و تست، و ساده‌تر شدن debug می‌شود.
- همیشه باید کدها را تا حد امکان مینیمال نگه داشت؛ کد کمتر = باگ کمتر = مسئولیت کمتر.
- در صورتی که نیاز دارید بعداً بخشی از کد را اضافه کنید، از version control مثل Git استفاده کنید؛ نگه‌داشتن کد مرده در سورس پروژه هیچ توجیهی ندارد.

پارت چهارم

۲.۶ اصل اجتناب از تکرار (Don’t Repeat Yourself - DRY)

اصل DRY می‌گه که نباید یک منطق یا اطلاعات تکراری توی کد به صورت کپی شده وجود داشته باشه. اگه یه بخش از کد قراره چند جا استفاده بشه، باید اون رو به یک تابع یا کلاس جدا تبدیل کنیم تا فقط یک بار تعریف بشه.

```
class Invoice:
    TAX_RATE = 0.09  # 9% tax

    def __init__(self, amount):
        self.amount = amount

    def calculate_tax(self):
        return self.amount * self.TAX_RATE

    def calculate_total(self):
        return self.amount + self.calculate_tax()
```

توضیح کامل:

۱. تو این مثال، محاسبه مالیات به‌صورت جداگانه تو تابع `calculate_tax` انجام شده و توی `calculate_total` از همون استفاده شده.

۲. اگه `self.amount * self.TAX_RATE` رو مستقیم توی `calculate_total` هم می‌نوشتیم، یعنی کد تکراری داشتیم و DRY نقض می‌شد.

۳. اصل DRY باعث می‌شه اگر لازم بود بعداً نرخ مالیات رو تغییر بدیم یا نحوه محاسبه مالیات تغییر کرد، فقط یه جا لازم باشه اصلاح کنیم.

۴. جلوگیری از تکرار، احتمال باگ رو کاهش می‌ده. اگه یه کد چند بار کپی شده باشه، ممکنه یکی از نسخه‌ها رو فراموش کنیم آپدیت کنیم.

۵. همچنین باعث بهبود خوانایی و تست‌پذیری می‌شه چون توابع یا کلاس‌ها هرکدوم مسئول انجام یک کار واحد هستند.

۶. DRY با اصل SRP (تک مسئولیت) هم‌جهته؛ هر کلاس یا تابع فقط باید مسئول یک منطق مشخص باشه.

۷. این اصل فقط به کدهای منطقی محدود نمی‌شه، حتی ثابت‌ها، رشته‌ها و مقادیر هم نباید چند بار تکرار بشن.

۸. توی پروژه‌های بزرگ، رعایت نکردن DRY باعث به وجود اومدن "technical debt" یا بدهی فنی می‌شه.

۹. بعضی وقتا توسعه‌دهنده‌ها چون عجله دارن، کد رو کپی‌پیست می‌کنن. این باعث می‌شه نگهداری کد به مرور سخت‌تر بشه.

۱۰. در نهایت، رعایت DRY باعث می‌شه تغییرات در سیستم سریع‌تر، کم‌خطاتر و قابل پیش‌بینی‌تر باشن.

۲.۷ اصل KISS (Keep It Simple, Stupid)

اصل KISS یعنی: **«ساده نگه‌دار!»** ایده‌اش اینه که: ساده‌ترین راه‌حل ممکن معمولاً بهترینه. نباید منطق‌های پیچیده، تو در تو، یا ساختارهای بیش‌ازحد طراحی‌شده بسازیم؛ مخصوصاً وقتی با یک روش ساده‌تر هم میشه همون کارو کرد.

```
# Bad: Overcomplicated logic
def is_even(number):
    if number % 2 == 0:
        return True
    else:
        return False

# Good: Keep it simple
def is_even(number):
    return number % 2 == 0
```

توضیح کامل:

۱. مثال بالا نشون می‌ده که چطور می‌تونیم یه منطق رو ساده‌تر و قابل خوندن‌تر کنیم بدون اینکه خروجی تغییر کنه.

۲. نسخه اول طولانی‌تره، تصمیم‌گیری (if/else) داره، و ذهن خواننده رو درگیر می‌کنه. در حالی که نسخه دوم خیلی راحت و بدون حاشیه جواب رو می‌ده.

۳. اصل KISS کمک می‌کنه تا توی تیم، کد قابل فهم‌تر و سریع‌تر قابل بررسی و تست باشه.

۴. وقتی کدی بیش از حد پیچیده می‌شه، احتمال خطا بالا می‌ره، حتی اگه از نظر منطقی درست باشه.

۵. سادگی همیشه به معنای بی‌کفایتی نیست؛ گاهی نوشتن یه راه‌حل ساده، نیاز به تجربه و درک عمیق‌تری داره.

۶. اصل KISS از این هم فراتر می‌ره و می‌گه که از طراحی‌های عجیب‌وغریب هم دوری کن. مثلاً لازم نیست برای یه لیست ساده، الگوریتم داده‌ساختار پیچیده پیاده‌سازی کنی.

۷. یکی از مهم‌ترین کاربردهای این اصل تو refactoring هست: ساده‌سازی منطق توابع و کلاس‌ها با حفظ عملکردشون.

۸. از نظر Clean Code، کدی که هر کسی (حتی تازه‌کار) بتونه سریع بفهمه چی کار می‌کنه، بهتر از یه کد "باهوشانه ولی گیج‌کننده" است.

۹. توی پروژه‌های واقعی، کد پیچیده‌تر یعنی زمان دیباگ و باگ‌گیری بیشتر.

۱۰. اصل KISS باعث می‌شه کار تیمی روان‌تر باشه چون بقیه افراد تیم سریع‌تر بتونن با کد ارتباط برقرار کنن.

پارت پنجم

۲.۸ اصل YAGNI (You Aren’t Gonna Need It)

اصل YAGNI یعنی: «**چیزی رو پیاده‌سازی نکن که الآن لازم نداری، حتی اگه فکر می‌کنی بعداً شاید به دردت بخوره.**»

```python
# Bad: Premature feature
class User:
    def __init__(self, username):
        self.username = username
        self.history = []  # not used anywhere

    def get_username(self):
        return self.username

    def save_history(self, command):
        self.history.append(command)  # this is not needed now
```

```python
# Good: Only what we need
class User:
    def __init__(self, username):
        self.username = username

    def get_username(self):
        return self.username
```

توضیح کامل:

۱. تو نسخه‌ی اول کلاس `User` ، یه قابلیت تاریخچه‌ی دستورات (`history`) اضافه شده که هنوز استفاده‌ای ازش نمی‌شه.

۲. این کار برخلاف اصل YAGNI هست، چون زمان و کدی برای چیزی صرف شده که نیاز فعلی سیستم نیست.

۳. خیلی وقت‌ها برنامه‌نویسا از روی پیش‌بینی نیازهای آینده، قابلیت‌هایی به سیستم اضافه می‌کنن که هیچ‌وقت استفاده نمی‌شن.

۴. این نه‌تنها زمان توسعه رو زیاد می‌کنه، بلکه کد رو پیچیده‌تر و نگهداری رو سخت‌تر می‌کنه.

۵. اگر بعداً واقعاً نیاز به `history` داشتیم، می‌تونیم به‌راحتی اون بخش رو اضافه کنیم، ولی الان اضافه کردنش هزینه و پیچیدگی اضافی داره.

۶. اصل YAGNI یکی از پایه‌های روش‌های توسعه چابک (Agile) هست که تأکید دارن فقط وقتی چیزی رو پیاده‌سازی کن که **واقعاً نیازه.**

۷. در پروژه‌های بزرگ، رعایت نکردن YAGNI باعث می‌شه بخشی از پروژه پر از فیچرهای ناقص، بی‌استفاده یا نیمه‌کاره باشه.

۸. کد اضافی باعث کاهش تمرکز و کیفیت کلی می‌شه، چون باید تست، دیباگ، و پشتیبانی هم بشه.

۹. حتی اگر یه فیچر بعداً لازم بشه، ممکنه اون موقع شرایط و نیازش فرق کرده باشه، پس کار قبلی‌مون ممکنه بی‌فایده باشه.

۱۰. تمرکز روی نیاز فعلی باعث می‌شه خروجی سریع‌تر، ساده‌تر و با کیفیت‌تر باشه.

۲.۹ اصل Code Should Explain Itself (کد باید خودش را توضیح دهد)

این اصل می‌گوید: **کدی بنویس که نیازی به توضیح اضافه نداشته باشه. خواننده با خوندن خود کد، بفهمه داره چی‌کار می‌کنه.**

```python
# Bad: Unclear and needs comments
def p(x):
    if x > 17:
        return True
    else:
        return False

# Good: Clear and self-explanatory
def is_adult(age):
    return age > 17
```

توضیح کامل:

۱. در نسخه‌ی اول، اسم تابع `p` و پارامتر `x` هیچ معنایی رو منتقل نمی‌کنن. فقط با خوندن کد متوجه نمی‌شیم منظور چیه.

۲. نسخه دوم با نام تابع `is_adult` و پارامتر `age` ، کاملاً روشن می‌کنه که این تابع بررسی می‌کنه فرد بزرگ‌ساله یا نه.

۳. وقتی نام توابع، متغیرها و کلاس‌ها گویا باشن، کمتر نیاز به نوشتن کامنت حس می‌شه.

۴. اصل "کد باید خودش رو توضیح بده" هم‌راستا با اصل Clean Code هست، چون خوانایی مهم‌تر از هوشمندی یا فشرده‌سازی بی‌رویه‌ی کده.

۵. برنامه‌نویسی تیمی جاییه که این اصل واقعاً می‌درخشه؛ چون توسعه‌دهنده‌های دیگه باید بدون پرس‌وجو بفهمن کد چی‌کار می‌کنه.

۶. این اصل کمک می‌کنه تا تست‌نویسی راحت‌تر باشه، چون وقتی تابع یا کلاس کارش روشنه، تست نوشتن برایش هم واضح‌تره.

۷. اگه مجبور شدی برای هر خط از کدت توضیح بنویسی، احتمال زیاد اسم‌ها یا ساختار تو بد تعریف شدن.

۸. البته این به معنی حذف کامنت‌های مفید نیست؛ بلکه یعنی اولویت با نوشتن کدی واضح و قابل فهمه.

۹. توابعی که کارهای پیچیده انجام می‌دن، بهتره به قسمت‌های کوچکتر تقسیم بشن تا هر کدوم عملکرد مشخصی داشته باشن.

۱۰. وقتی یه کلاس یا تابع بدون توضیح اضافی خونده می‌شه و منظورش قابل درکه، یعنی داری یه Clean Code واقعی می‌نویسی.

پارت ششم

۲.۱۰ اصل Avoid Magic Numbers (از اعداد جادویی پرهیز کن)

اعداد جادویی (Magic Numbers) به اعدادی گفته می‌شود که در کد به صورت مستقیم استفاده می‌شوند بدون اینکه دلیل یا معنای مشخصی داشته باشند. این اعداد معمولاً باعث سردرگمی می‌شوند و اگر نیاز به تغییر اون‌ها باشه، باید در همه‌جا جستجو بشن. به‌جای استفاده از اعداد جادویی، بهتره از ثابت‌ها (constants) یا متغیرهای معنادار استفاده کنیم.

```python
# Bad: Magic Number
def calculate_area(radius):
    return 3.14159 * radius * radius

# Good: Avoid Magic Numbers
PI = 3.14159
def calculate_area(radius):
    return PI * radius * radius
```

توضیح کامل:

۱. در نسخه‌ی اول، عدد `3.14159` که همان π است، به‌صورت مستقیم در کد استفاده شده است. اگر این عدد در چندین نقطه از کد تکرار بشه، می‌تواند باعث مشکلاتی مثل اشتباه در تغییر مقدار یا درک سخت از مفهوم آن شود.

۲. در نسخه‌ی دوم، این عدد به یک ثابت `PI` تغییر داده شده که می‌توان آن را در یک جای واحد مدیریت و تغییر داد.

۳. وقتی از اعداد جادویی استفاده می‌کنیم، فهمیدن دلیل استفاده از آن‌ها سخت می‌شود، اما با تعریف ثابت‌ها یا متغیرهای معنادار، کد خود به خود قابل فهم‌تر می‌شود.

۴. استفاده از ثابت‌ها باعث می‌شود که تغییرات لازم در یک جا اعمال شوند، بنابراین کد کمتر در معرض خطا قرار می‌گیرد.

۵. در پروژه‌های بزرگ، استفاده از ثابت‌ها به‌ویژه برای مقادیری مثل نرخ مالیات، ضریب‌ها و سایر اعداد شناخته‌شده، نگهداری کد را راحت‌تر می‌کند.

۶. اعداد جادویی باعث افزایش احتمال بروز اشتباهات می‌شود، به خصوص زمانی که پروژه گسترش می‌یابد و نیاز به تغییرات زیادی دارد.

۷. این اصل همچنین به بهبود تست‌پذیری کمک می‌کند، چون می‌توانیم ثابت‌ها را تست کرده و آن‌ها را در تست‌های مختلف تغییر دهیم.

۸. اعداد جادویی در نهایت از خوانایی کد می‌کاهند. وقتی کسی کد را می‌خواند، باید بتواند فوراً بفهمد که این عدد چه معنایی دارد.

۹. حتی اگر کدی برای مدت طولانی نیاز به تغییر نداشته باشد، استفاده از ثابت‌ها برای این اعداد تضمین می‌کند که در آینده راحت‌تر نگهداری می‌شود.

۱۰. در زبان‌های برنامه‌نویسی‌ای مثل Python، استفاده از ثابت‌ها نه‌تنها از نظر بهبود خوانایی، بلکه از نظر جلوگیری از اشتباهات ناخواسته در طول پروژه‌های بزرگ بسیار مفید است.

۲.۱۱ اصل حداقل وابستگی (Minimize Dependencies)

اصل حداقل وابستگی به این معناست که هر واحد کد (کلاس، ماژول یا تابع) باید حداقل وابستگی را به دیگر واحدها داشته باشد. این اصل برای کاهش پیچیدگی و افزایش انعطاف‌پذیری کد اهمیت زیادی دارد. هرچه وابستگی‌ها بیشتر شوند، تغییرات در یک بخش از کد ممکن است باعث تغییرات غیرمنتظره در بخش‌های دیگر شود.

```
# Bad: High Dependency
class Report:
    def __init__(self, data):
        self.data = data

    def generate_pdf(self):
        # Generates PDF report (specific dependency)
        pass

    def generate_csv(self):
        # Generates CSV report (specific dependency)
        pass

# Good: Low Dependency
class Report:
```

```
    def __init__(self, data, generator):
        self.data = data
        self.generator = generator  # Dependency injection

    def generate(self):
        self.generator.generate(self.data)
```

توضیح کامل:

۱. در نسخه اول، کلاس `Report` به طور مستقیم به نحوه تولید گزارش (PDF و CSV) وابسته است. این یعنی اگر بخواهیم نوع دیگری از گزارش تولید کنیم، باید این کلاس را تغییر دهیم.

۲. در نسخه دوم، از مفهوم **تزریق وابستگی (Dependency Injection)** استفاده شده است. در اینجا، کلاس `Report` فقط مسئول ذخیره داده‌ها و فراخوانی گزارش‌دهنده (generator) است، و خود به خود نحوه تولید گزارش را پیاده‌سازی نمی‌کند.

۳. این نوع طراحی وابستگی کمتری ایجاد می‌کند و باعث می‌شود که تغییرات در نحوه تولید گزارش (مثلاً اضافه کردن خروجی Excel) نیاز به تغییر در کلاس `Report` نداشته باشد.

۴. **تزریق وابستگی** یک روش موثر برای کاهش وابستگی است که به ما اجازه می‌دهد از کلاس‌های مختلف برای تولید انواع مختلف گزارش‌ها استفاده کنیم بدون اینکه به کد اصلی وابسته باشیم.

۵. این طراحی باعث می‌شود که کلاس `Report` قابل تست‌تر شود، چون به جای وابستگی به یک پیاده‌سازی خاص، می‌توانیم هر نوع پیاده‌سازی دلخواه را به آن تزریق کنیم.

۶. این اصل از پیچیدگی جلوگیری می‌کند و وقتی سیستم بزرگتر می‌شود، باعث می‌شود که تغییرات در بخش‌های مختلف کد اثرات جانبی کمتری داشته باشند.

۷. با کاهش وابستگی‌ها، کد انعطاف‌پذیرتر و ماژولارتر می‌شود و توسعه‌دهنده‌ها می‌توانند بخش‌های مختلف را جداگانه تغییر دهند و تست کنند.

۸. استفاده از تزریق وابستگی باعث می‌شود که کد به راحتی قابل گسترش باشد، چون کلاس‌ها می‌توانند به راحتی از هم جدا شوند و تغییرات در یکی، دیگری را تحت تاثیر قرار ندهد.

۹. این اصل همچنین به ترویج طراحی‌های بهتر و معماری‌های ماژولار کمک می‌کند که در پروژه‌های بزرگ‌تر کاملاً ضروری است.

۱۰. در نهایت، کاهش وابستگی‌ها باعث بهبود قابل تست بودن، نگهداری آسان‌تر و کاهش پیچیدگی سیستم می‌شود.

۲.۱۲ تست‌پذیری بالا (High Testability)

تست‌پذیری بالا یعنی کدی بنویسیم که به راحتی قابل تست و بررسی باشد. وقتی کد به صورت ماژولار و با جداکردن مسئولیت‌ها طراحی می‌شود، تست آن راحت‌تر خواهد بود. تست‌پذیری بالا به این معناست که هر بخش از کد به‌طور مستقل قابل آزمایش است و تغییرات در یک بخش، تأثیری روی بقیه قسمت‌ها ندارد.

```python
# Bad: Low Testability
class Payment:
    def process_payment(self, amount, card_number):
        # complex logic here
        pass

    def send_email_confirmation(self, email):
        # email sending logic
        pass

# Good: High Testability (Separation of Concerns)
class Payment:
    def __init__(self, processor, notifier):
        self.processor = processor  # Dependency Injection
        self.notifier = notifier

    def process_payment(self, amount, card_number):
        result = self.processor.process(amount, card_number)
        self.notifier.send_confirmation(result)
        return result
```

توضیح کامل:

۱. در نسخه‌ی اول، کلاس `Payment` هم مسئول پردازش پرداخت‌ها است و هم مسئول ارسال تاییدیه‌های ایمیل.

۲. این طراحی باعث می‌شود که تست هر یک از این وظایف به‌طور جداگانه سخت باشد و وابستگی‌ها پیچیده شوند.

۳. در نسخه‌ی دوم، با استفاده از **تزریق وابستگی (Dependency Injection)**، مسئولیت‌ها از هم جدا شده‌اند. کلاس `Payment` فقط به یک `processor` برای پردازش و یک `notifier` برای ارسال ایمیل

نیاز دارد.

۴. این طراحی باعث می‌شود که هر یک از بخش‌ها به‌طور مستقل قابل تست باشند. یعنی می‌توانیم ابتدا پردازش پرداخت را آزمایش کنیم، سپس تاییدیه ایمیل را.

۵. این اصل همچنین از آنجایی که تست‌ها به راحتی می‌توانند وابستگی‌ها را شبیه‌سازی کنند (Mock)، تست‌پذیری را بسیار بهبود می‌دهد.

۶. وقتی هر واحد به‌طور مستقل قابل تست باشد، خطاها سریع‌تر شناسایی و اصلاح می‌شوند.

۷. با تست‌پذیری بالا، می‌توانیم به سرعت تغییرات را در کد اعمال کنیم و مطمئن شویم که هیچ‌کدام از ویژگی‌های دیگر به اشتباه تغییر نکرده‌اند.

۸. همچنین تست‌پذیری بالا باعث می‌شود که کد قابل نگهداری‌تر باشد، چرا که با داشتن مجموعه‌ای از تست‌های خودکار می‌توانیم به‌راحتی به صحت عملکرد کد اطمینان حاصل کنیم.

۹. در پروژه‌های بزرگ، تست‌پذیری بالا به طرز چشمگیری زمان توسعه و نگهداری سیستم را کاهش می‌دهد.

۱۰. اگر کد به درستی تست‌شده باشد، در صورت نیاز به گسترش یا اصلاح آن، می‌توانیم اطمینان حاصل کنیم که سیستم همچنان به درستی کار می‌کند.

پارت هفتم

۲.۱۳ کاهش پیچیدگی (Reduce Complexity)

کاهش پیچیدگی یکی از اصول اساسی در نوشتن کد تمیز است. پیچیدگی کد می‌تواند در هنگام گسترش سیستم باعث بروز مشکلاتی مانند افزایش زمان نگهداری، سختی در افزودن ویژگی‌های جدید و ایجاد خطاهای غیرمنتظره شود. برای کاهش پیچیدگی، باید کد را ساده، ماژولار و قابل فهم نگه داریم.

```python
# Bad: High Complexity
def process_order(order):
    if order.status == 'new':
        if order.customer.is_vip():
            if order.total_price > 100:
                process_vip_order(order)
            else:
                process_regular_order(order)
        else:
            process_regular_order(order)
    else:
        print("Order already processed")

# Good: Reduced Complexity (Simplified with separate functions)
def process_order(order):
    if order.status != 'new':
        print("Order already processed")
        return

    if order.customer.is_vip():
        process_vip_order(order)
    else:
        process_regular_order(order)
```

توضیح کامل:

۱. در نسخه‌ی اول، تابع `process_order` بسیار پیچیده است و با چندین شرط تو در تو و بررسی‌های مختلف روبه‌رو می‌شویم.
۲. این پیچیدگی باعث می‌شود که کد به سختی خوانده شود و مدیریت تغییرات در آن دشوار باشد.
۳. در نسخه‌ی دوم، پیچیدگی با تفکیک بخش‌های مختلف کد کاهش پیدا کرده است. ابتدا بررسی می‌کنیم که آیا سفارش جدید است یا نه، سپس بررسی می‌کنیم که آیا مشتری VIP است یا نه.
۴. با این کار، تابع `process_order` ساده‌تر و قابل فهم‌تر شده و وظیفه هر بخش به‌وضوح مشخص است.
۵. کاهش پیچیدگی نه‌تنها خوانایی را بهبود می‌دهد بلکه باعث می‌شود که کد به راحتی قابل تست و نگهداری باشد.
۶. پیچیدگی زیاد می‌تواند باعث افزایش زمان اجرای کد شود، اما وقتی که کد ساده و خوانا باشد، به راحتی قابل بهینه‌سازی است.
۷. وقتی تعداد شرایط یا متغیرهای داخلی زیاد شود، کد پیچیده‌تر می‌شود و به راحتی دچار اشتباهات ناخواسته می‌شود.
۸. با ساده کردن منطق برنامه، حتی وقتی نیاز به اضافه کردن ویژگی‌های جدید باشد، تغییرات راحت‌تر و با احتمال خطای کمتری انجام می‌شود.
۹. همچنین این نوع طراحی باعث می‌شود که هنگام بررسی یا دیباگ کردن کد، فهمیدن علت مشکلات بسیار راحت‌تر باشد.
۱۰. در نهایت، کاهش پیچیدگی به معنی ساده‌سازی روند توسعه است. این کار باعث می‌شود تیم‌های توسعه بتوانند سریع‌تر و با کیفیت بالاتری کد بنویسند.

۲.۱۴ کد تکراری را حذف کن (Eliminate Duplicate Code)

حذف کد تکراری یکی از اصول کلیدی در کدنویسی تمیز است. کد تکراری نه تنها باعث می‌شود که نگهداری کد مشکل‌تر شود، بلکه می‌تواند منجر به ایجاد خطاهای غیرمنتظره و پیچیدگی در پروژه‌های بزرگ شود. یکی از اهداف اصلی این اصل، استفاده از **DRY** (Don't Repeat Yourself) است که به معنای "از تکرار خودداری کن" می‌باشد.

```
# Bad: Duplicate Code
class Report:
    def generate_pdf(self, data):

```

```
        # Logic to generate PDF
        pass

    def generate_csv(self, data):
        # Logic to generate CSV
        pass

# Good: Eliminating Duplicate Code
class ReportGenerator:
    def generate(self, data, format):
        if format == 'pdf':
            self.generate_pdf(data)
        elif format == 'csv':
            self.generate_csv(data)

    def generate_pdf(self, data):
        # Logic to generate PDF
        pass

    def generate_csv(self, data):
        # Logic to generate CSV
        pass
```

توضیح کامل:

۱. در نسخه اول، کد برای تولید گزارش در دو فرمت مختلف (PDF و CSV) تکرار شده است. این باعث می‌شود که اگر بخواهیم تغییراتی در نحوه تولید گزارش‌ها ایجاد کنیم، باید در هر دو متد `generate_pdf` و `generate_csv` تغییرات مشابهی اعمال کنیم.

۲. در نسخه دوم، منطق تولید گزارش در یک مکان (کلاس `ReportGenerator`) متمرکز شده است و با استفاده از متد `generate` ، فرمت گزارش تعیین می‌شود. این باعث می‌شود که هیچ‌گونه تکرار کد در این بخش وجود نداشته باشد.

۳. با این روش، وقتی نیاز به افزودن فرمت جدیدی برای گزارش داشته باشیم، فقط کافیست یک شرط دیگر در متد `generate` اضافه کنیم بدون اینکه مجبور به تغییر در منطق تولید هر فرمت شویم.

۴. حذف کد تکراری موجب ساده‌تر شدن نگهداری کد می‌شود، چون اگر بخواهیم به‌روزرسانی‌هایی در فرآیند تولید گزارش‌ها انجام دهیم، فقط یک بار آن را اعمال می‌کنیم.

۵. این اصل همچنین به جلوگیری از بروز خطاهای احتمالی کمک می‌کند. وقتی کد تکراری وجود دارد، احتمال این که به طور تصادفی یک مورد را فراموش کنیم یا اشتباهی مشابه را دوباره در جای دیگر کد ایجاد کنیم، بیشتر می‌شود.
۶. از آنجایی که کد تکراری باعث پیچیدگی در فهم کد می‌شود، حذف آن خوانایی و فهم کد را بهبود می‌دهد.
۷. این اصل به ویژه در پروژه‌های بزرگ مفید است. در چنین پروژه‌هایی ممکن است چندین بخش از کد نیاز به انجام کار مشابهی داشته باشند و این می‌تواند باعث گسترش مشکلات در سراسر پروژه شود.
۸. حذف کد تکراری همچنین باعث کاهش حجم کد می‌شود، که به راحتی آن را قابل مدیریت‌تر می‌کند.
۹. **DRY** نه تنها کد را تمیزتر می‌کند، بلکه باعث می‌شود که تست‌پذیری سیستم نیز بهتر شود، چرا که با داشتن یک منبع واحد برای هر منطق، تنها باید آن را یک بار تست کرد.
۱۰. در نهایت، این اصل باعث بهبود کیفیت نرم‌افزار و کاهش زمان نگهداری آن می‌شود. کد بدون تکرار، پایدارتر و توسعه‌پذیرتر است.

پارت هشتم

۳.۱ اصل Single Responsibility Principle (SRP)

اصل **Single Responsibility Principle (SRP)** می‌گوید که هر کلاس باید تنها یک مسئولیت داشته باشد. یعنی، هر کلاس باید برای یک هدف خاص طراحی شود و تنها به دلیل تغییر در آن مسئولیت تغییر کند. این اصل کمک می‌کند که کد ما قابل نگهداری‌تر، تست‌پذیرتر و انعطاف‌پذیرتر باشد.

```python
class Invoice:
    def __init__(self, amount, customer):
        self.amount = amount
        self.customer = customer

    def calculate_tax(self):
        return self.amount * 0.1

    def save_to_database(self):
        # Simulate saving to database
        print(f"Saving invoice for {self.customer} to database.")
```

توضیح کامل:

- در اینجا کلاس `Invoice` دو مسئولیت مختلف دارد: یکی برای محاسبه مالیات و دیگری برای ذخیره‌سازی اطلاعات در پایگاه داده. این به این معناست که اگر نیاز به تغییر نحوه‌ی محاسبه مالیات یا نحوه‌ی ذخیره‌سازی داده‌ها داشته باشیم، باید در این کلاس تغییراتی ایجاد کنیم.
- طبق اصل SRP، باید این دو مسئولیت را از هم جدا کنیم. به این ترتیب، اگر نیاز به تغییر یکی از این وظایف داشته باشیم، تنها باید کلاسی را تغییر دهیم که مسئول آن وظیفه است و نه کلاسی که مسئول وظیفه دیگر است.
- برای حل این مشکل، می‌توانیم یک کلاس جدید به نام `DatabaseSaver` بسازیم که تنها مسئول ذخیره‌سازی داده‌ها باشد، و کلاسی که مسئول محاسبه مالیات است را از آن جدا کنیم.
- این کار باعث می‌شود که کد شما ساده‌تر، قابل تست‌تر و قابل توسعه‌تر شود. به‌علاوه، مسئولیت‌ها به‌طور جداگانه نگهداری می‌شوند، بنابراین کد به راحتی قابل تغییر است.

- اگر با رعایت SRP، کد را به واحدهای کوچکتر تقسیم کنیم، دیگر هیچ یک از اجزای سیستم به اجبار درگیر تغییرات در سایر بخش‌ها نخواهند شد.
- این اصل به کاهش پیچیدگی کد کمک می‌کند و باعث می‌شود که هر کلاس تنها به یک هدف خاص پرداخته و دیگر نیازی به تغییرات بی‌رویه در بخش‌های دیگر نخواهد بود.
- پیروی از SRP همچنین به نگهداری و مدیریت کد در تیم‌های بزرگ کمک می‌کند زیرا هر کسی می‌تواند به‌راحتی بر روی بخشی از کد تمرکز کند بدون اینکه نگران اثرات جانبی تغییرات در بخش‌های دیگر باشد.
- این اصل به شما این امکان را می‌دهد که برای هر بخش از سیستم واحدهای خاصی از کد ایجاد کنید که تنها به وظیفه خاص خود پرداخته و نیازی به تغییرات غیرضروری نداشته باشند.
- در نهایت، با رعایت SRP، کد شما مقیاس‌پذیرتر و قابل فهم‌تر خواهد شد، زیرا هر کلاس برای یک وظیفه خاص طراحی شده است و توسعه‌دهندگان به‌راحتی می‌توانند مسئولیت‌های آن را درک کنند.

۳.۲ اصل Open/Closed Principle (OCP)

اصل **Open/Closed Principle (OCP)** می‌گوید که کلاس‌ها باید برای گسترش باز و برای تغییر بسته باشند. به این معنی که شما باید بتوانید ویژگی‌های جدیدی به سیستم اضافه کنید بدون اینکه کد موجود را تغییر دهید.

```
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height


```

```
    def area(self):
        return self.width * self.height
```

توضیح کامل:

در این مثال، کلاس Shape یک متد area دارد که توسط زیرکلاس‌ها پیاده‌سازی می‌شود. این امکان به ما می‌دهد که هر شکل جدیدی (مثل Triangle) به سیستم اضافه کنیم بدون اینکه بخواهیم تغییرات زیادی در کدهای قبلی ایجاد کنیم.

به‌طور خاص، اگر بخواهیم شکل‌های جدیدی به سیستم اضافه کنیم، فقط باید کلاس‌های جدیدی تعریف کنیم که متد area را پیاده‌سازی کنند و نیازی به تغییر در کد کلاس‌های قبلی مانند Circle و Rectangle نیست.

طبق اصل OCP، این طراحی باعث می‌شود که کد برای تغییرات آینده آماده باشد. به‌جای اینکه به طور مداوم کد موجود را تغییر دهیم، می‌توانیم به راحتی ویژگی‌های جدیدی را اضافه کنیم.

این اصل کمک می‌کند که سیستم شما برای تغییرات باز باشد بدون اینکه درگیر کدهای موجود شوید، بنابراین باعث کاهش احتمال بروز اشکالات ناشی از تغییرات در کد می‌شود.

پیروی از OCP به توسعه‌دهندگان این امکان را می‌دهد که قابلیت‌های جدیدی به سیستم اضافه کنند و در عین حال کد قبلی را دست‌نخورده و سالم نگه دارند.

برای پیاده‌سازی OCP، معمولاً از وراثت یا ترکیب استفاده می‌شود تا به شما اجازه دهد که کلاس‌های جدید را بدون تغییر در کلاس‌های قبلی اضافه کنید.

این اصل به شما کمک می‌کند که طراحی سیستم‌های مقیاس‌پذیر و قابل توسعه داشته باشید و از تغییرات پیچیده در کدهای اصلی جلوگیری کنید.

در صورتی که نیاز به افزودن قابلیت‌های جدید باشد، این اصل از ایجاد تغییرات غیرضروری در کلاس‌های پایه جلوگیری می‌کند و سیستم به‌طور خودکار قادر به گسترش خواهد بود.

این اصل می‌تواند به شما کمک کند تا کدهایی بنویسید که در برابر تغییرات مقاوم هستند و از افزایش پیچیدگی و خطا در پروژه‌های بزرگ جلوگیری می‌شود.

با استفاده از OCP، سیستم شما قابلیت توسعه و ارتقاء آسانی خواهد داشت بدون اینکه به قسمت‌های قبلی سیستم آسیب برسد.

۳.۳ اصل Liskov Substitution Principle (LSP)

اصل **Liskov Substitution Principle (LSP)** می‌گوید که هر زیرکلاسی باید بتواند به‌طور جایگزین برای کلاس پایه خود استفاده شود، بدون اینکه رفتار نامطلوبی ایجاد کند.

```
class Bird:
    def fly(self):
        print('Flying')

class Sparrow(Bird):
    pass  # Sparrow can fly

class Penguin(Bird):
    def fly(self):
        raise NotImplementedError('Penguins cannot fly')  # Penguins canr
```

توضیح کامل:

- در این مثال، کلاس `Bird` یک متد `fly` دارد که تمام پرندگان باید بتوانند آن را پیاده‌سازی کنند. در حالت معمول، این متد باید در تمام زیرکلاس‌ها رفتار یکسانی داشته باشد.
- اما پنگوئن به‌طور ذاتی نمی‌تواند پرواز کند، بنابراین پیاده‌سازی متد `fly` در کلاس `Penguin` باعث ایجاد خطا می‌شود و این با اصل Liskov Substitution Principle مغایرت دارد.
- طبق این اصل، هر زیرکلاس باید قادر باشد به‌طور جایگزین در جایگاه کلاس پایه استفاده شود بدون اینکه باعث بروز خطا یا تغییر رفتار ناخواسته در کد شود.
- برای رفع این مشکل، باید طراحی را به گونه‌ای انجام دهیم که این توقع از تمام پرندگان نداشته باشیم که همه‌شان می‌توانند پرواز کنند. به این ترتیب می‌توانیم دو دسته از پرندگان تعریف کنیم: پرندگانی که می‌توانند پرواز کنند و پرندگانی که نمی‌توانند.
- برای این کار می‌توانیم از چندین کلاس انتزاعی (مانند `FlyingBird` و `NonFlyingBird`) استفاده کنیم تا از این اشتباه جلوگیری کنیم و پیاده‌سازی‌های متفاوت برای پرندگان مختلف ارائه دهیم.

- این اصل اهمیت زیادی دارد، زیرا به شما کمک می‌کند که سلسله‌مراتب‌های وراثتی معقول و معنی‌دار ایجاد کنید که کاملاً با رفتارهای کلاس پایه تطابق داشته باشد.
- رعایت LSP به شما این امکان را می‌دهد که به راحتی از زیرکلاس‌ها استفاده کنید و اطمینان حاصل کنید که هیچ رفتار غیرمنتظره‌ای در سیستم ایجاد نمی‌شود.
- این اصل همچنین کمک می‌کند که سیستم‌های پیچیده‌تر بدون بروز مشکلات خاصی در کد به‌طور کارآمدتر و منعطف‌تر عمل کنند.
- در نهایت، پیروی از LSP باعث می‌شود که کد شما قابل پیش‌بینی و ایمن‌تر باشد و کدهای شما در برابر تغییرات مقاوم‌تر خواهند بود.

پارت نهم

۳.۴ اصل Interface Segregation Principle (ISP)

اصل **Interface Segregation Principle (ISP)** می‌گوید که بهتر است یک رابط بزرگ و جامع به رابط‌های کوچک‌تر و خاص‌تر تقسیم شود. به عبارت دیگر، هر کلاس یا ماژول نباید مجبور به پیاده‌سازی متدهایی شود که به آن‌ها نیازی ندارد.

```python
class Printer:
    def print_document(self, document):
        print(f"Printing document: {document}")

    def scan_document(self, document):
        raise NotImplementedError("Scanner functionality not implemented'

class Scanner:
    def scan_document(self, document):
        print(f"Scanning document: {document}")

class MultiFunctionPrinter(Printer, Scanner):
    def scan_document(self, document):
        print(f"Scanning document in multifunction printer: {document}")

    def print_document(self, document):
        print(f"Printing document in multifunction printer: {document}")
```

توضیح کامل:

در این مثال، کلاس Printer دارای یک متد print_document است، اما متد scan_document را پیاده‌سازی نکرده است. این به این معناست که یک پرینتر که به خودی خود قابلیت اسکن ندارد، مجبور به پیاده‌سازی متد scan_document می‌شود، که با اصل ISP مغایر است.

طبق اصل ISP، باید از پیاده‌سازی رابط‌های بزرگ که شامل متدهای غیرضروری هستند پرهیز کرد. بهتر است از رابط‌های کوچکتر و خاص‌تر استفاده کنیم که تنها مسئولیت‌های مرتبط با خود را پوشش دهند.

به همین دلیل، یک پرینتر که فقط قابلیت چاپ دارد نباید مجبور باشد متد اسکن را پیاده‌سازی کند. برای این کار می‌توانیم رابط‌های جداگانه‌ای برای چاپ و اسکن تعریف کنیم.

کلاسی مانند MultiFunctionPrinter که هر دو قابلیت چاپ و اسکن را پیاده‌سازی می‌کند، به درستی به این اصل پایبند است چون تنها زمانی متدهای مربوط به چاپ و اسکن را پیاده‌سازی می‌کند که نیاز به هر دو قابلیت وجود داشته باشد.

استفاده از این طراحی باعث می‌شود که سیستم شما انعطاف‌پذیرتر و قابل گسترش‌تر باشد. برای مثال، اگر بخواهیم یک کلاس جدید برای دستگاهی بنویسیم که تنها چاپ می‌کند و اسکن نمی‌کند، به راحتی می‌توانیم تنها رابط Printer را پیاده‌سازی کنیم.

مفهوم ISP به کاهش پیچیدگی سیستم کمک می‌کند، زیرا باعث می‌شود کلاس‌ها تنها وظایف خاصی را پیاده‌سازی کنند و از کدهای اضافی و پیچیده جلوگیری شود.

این اصل به‌ویژه در سیستم‌های بزرگ و پیچیده مفید است، جایی که با تقسیم وظایف به بخش‌های کوچکتر، نگهداری و توسعه کد راحت‌تر می‌شود.

۳.۵ اصل Dependency Inversion Principle (DIP)

اصل **Dependency Inversion Principle (DIP)** می‌گوید که ماژول‌های سطح بالا نباید به ماژول‌های سطح پایین وابسته باشند، بلکه باید به انتزاع‌ها وابسته باشند. همچنین، انتزاع‌ها نباید به جزئیات وابسته باشند، بلکه جزئیات باید به انتزاع‌ها وابسته باشند. به عبارت ساده‌تر، شما باید از وابستگی‌های مستقیم به کلاس‌های خاص پرهیز کنید و از وابستگی به انتزاع‌ها استفاده کنید.

```
class DatabaseConnection:
    def connect(self):
        print("Connecting to the database...")

class DataService:
    def __init__(self, connection: DatabaseConnection):
        self.connection = connection

    def fetch_data(self):

```

```
        self.connection.connect()
        print("Fetching data from the database.")

# Dependency injection through constructor
db_connection = DatabaseConnection()
data_service = DataService(db_connection)
data_service.fetch_data()
```

توضیح کامل:

در این مثال، کلاس DataService به کلاس DatabaseConnection وابسته است. طبق DIP، این نوع وابستگی به کلاس‌های سطح پایین مشکلی ایجاد می‌کند زیرا DataService به‌طور مستقیم به یک پیاده‌سازی خاص (DatabaseConnection) وابسته است.

طبق اصل DIP، باید از وابستگی به انتزاع‌ها (مثل رابط‌ها یا کلاس‌های پایه) استفاده کرد، نه پیاده‌سازی‌های خاص. در این حالت، DataService باید به یک انتزاع مانند DatabaseConnectionInterface وابسته باشد، نه به یک کلاس خاص.

این کار باعث می‌شود که کلاس DataService از پیاده‌سازی‌های مختلف دیتابیس (مثل MongoDB، MySQL یا SQLite) پشتیبانی کند، بدون اینکه نیاز به تغییر در کد DataService داشته باشیم.

به‌طور خاص، کلاس‌های سطح پایین (مثل DatabaseConnection) نباید به‌طور مستقیم در کلاس‌های سطح بالا (مثل DataService) استفاده شوند، بلکه باید از طریق وابستگی به انتزاع‌ها انجام شود.

برای پیاده‌سازی این اصل، می‌توان از dependency injection استفاده کرد که به این معناست که وابستگی‌های لازم به یک کلاس از بیرون آن کلاس تزریق می‌شود.

این کار باعث می‌شود که کد شما انعطاف‌پذیرتر باشد و بتوانید به راحتی جزئیات مختلف (مثل انواع مختلف پایگاه داده) را تغییر دهید بدون اینکه در کدهای سطح بالا تغییری ایجاد کنید.

همچنین استفاده از این روش باعث می‌شود که تست کردن کد ساده‌تر باشد، زیرا می‌توانید به‌راحتی پیاده‌سازی‌های مختلف را به جای کلاس‌های واقعی تزریق کنید.

مفهوم DIP کمک می‌کند که سیستم شما مقیاس‌پذیرتر و قابل تغییر باشد، زیرا تغییرات در پیاده‌سازی‌ها باعث تغییرات در کلاس‌های سطح بالا نمی‌شود.

در نتیجه، این اصل از ایجاد وابستگی‌های مستقیم بین کلاس‌ها جلوگیری می‌کند و از این طریق سیستم شما را از تغییرات ناخواسته محافظت می‌کند.