

۱۰ اصل طلایی Clean Code

چکیده‌ای از کتاب Clean Code | نوشته‌ی Robert C. Martin



اگر این نوشته را می‌خوانی، دو دلیل بیشتر ندارد: اول این که برنامه‌نویسی. دوم این که می‌خواهی برنامه‌نویس بهتری شوی. **خوش آمدی!**

دنیا به برنامه‌نویس‌های بیشتری نیاز ندارد؛ به برنامه‌نویس‌های بهتری نیاز دارد.

در ادامه چکیده‌ی یکی از تأثیرگذارترین کتاب‌های تاریخ برنامه‌نویسی را می‌خوانی:

Clean Code نوشته‌ی Robert C. Martin — راهنمایی برای نوشتن کدی که فقط کار نمی‌کند، بلکه قابل فهم، قابل نگهداری و قابل احترام است.

اصل ۱: نام‌ها باید معنی داشته باشند



هیچ چیز مثل یک نام بد نمی‌تواند کد را آزاردهنده کند.

اسم اشتباه، مثل برچسب اشتباه روی داروست؛ همه چیز درست کار می‌کند، تا وقتی که یکی بخواهد از آن استفاده کند!

برنامه‌نویس حرفه‌ای وقت می‌گذارد تا اسم خوب پیدا کند. نه از روی وسواس، بلکه از روی احترام — احترام به ذهن کسی که بعداً این کد را می‌خواند، حتی اگر آن شخص، نسخه‌ی آینده‌ی خود او باشد.

چگونه نام‌گذاری حرفه‌ای انجام دهیم

۱. اسم باید بگوید چه می‌کند، نه فقط چه هست.

○ تابع `getUserData()` بهتر از `handleUser()` است، چون هدف را مشخص می‌کند.

۲. از کلمات عمومی و مبهم مثل `info`, `data`, `manager`, `process` دوری کن.
۳. طول نام مهم نیست، وضوح مهم است.
نام بلند و واضح، بهتر از نام کوتاه اما مبهم هست.
`calculateInvoiceTotal()` بهتر از `calcInv()` است.
۴. برای تابع از فعل استفاده کن، برای متغیر از اسم.
تابع باید عمل را توصیف کند (`sendEmail()`)،
متغیر باید چیزی را نشان دهد (`userEmail`).
۵. از نام‌های نادرست یا دروغ‌گو پرهیز کن.
۶. از تفاوت ظریف حروف یا اختصار برای تفاوت مفهومی استفاده نکن.
○ `accountList` و `accountData` یعنی سردرگمی قطعی.
۷. نام‌ها باید در یک دامنه، یک‌دست باشند.
○ اگر یک‌جا `fetch` می‌نویسی، جای دیگر `get` ننویس. انسجام، نشانه‌ی تفکر طراحی است
۸. از شوخی و ابهام دوری کن.
○ کد، محل نبوغ ادبی نیست. نامی که باعث لبخند شود، شاید باعث خطا هم بشود.

اصل ۲: تابع باید فقط یک کار انجام دهد

تابع تمیز باید یک کار انجام دهد. فقط یکی.

هرچه تعداد خطوط بیشتر شود، احتمال دروغ‌گویی تابع درباره‌ی نیتش بیشتر است.
تو باید بتوانی با یک نگاه بگویی این تابع دقیقاً چه می‌کند — نه اینکه حدس بزنی.



چگونه تابع تمیز بنویسیم

۱. تابع باید کوچک باشد: اغلب ۵ تا ۱۵ خط کافی است.
۲. هر تابع فقط یک کار انجام دهد: اگر در توضیحش مجبور شدی از "و" استفاده کنی، یعنی چندکار می‌کند: «ذخیره و ارسال و ثبت در لاگ» = سه تابع.
۳. نام تابع باید دقیق بگوید چه می‌کند: نام دقیق، خودش تست اولیه‌ی صحت منطق است `sendWelcomeEmail()`. بهتر از `processUser()` است.
۴. تابع باید فقط در یک «سطح فکر» صحبت کند: یا درباره‌ی چپستی کار حرف بزن (سطح بالا)، یا درباره‌ی چگونگی انجامش (سطح پایین) — نه هر دو با هم.
۵. تعداد پارامترها را حداقل نگه دار: دو پارامتر قابل قبول است؛ سه مورد مرزی است؛ بیشتر از سه یعنی نیاز به بازطراحی داری.

۶. خروجی واضح، ورودی شفاف: تابع نباید چیزی را در سکوت تغییر دهد (مثلاً state بیرونی). (هر چه انجام می‌دهد باید از نام و امضایش معلوم باشد).
۷. از شرط‌های تودرتو پرهیز کن: هر if یا switch باید راه خروج ساده و واضح داشته باشد. کد باید مثل مسیر مستقیم خوانده شود، نه مثل هزارتو.
۸. تابع‌ها را طوری بنویس که بتوانی آن‌ها را با هم بخوانی: وقتی توابع پشت سرهم نوشته می‌شوند، باید مثل روایت طبیعی خوانده شوند:

○ ○ ○

```
checkPermission();  
validateInput();  
saveUser();  
sendEmail();
```

اصل ۳: کامنت، جایگزین کد بد نیست



کامنت خوب مثل چسب زخم است؛ گاهی لازم است، اما اگر زیاد شد یعنی زخم عمیق‌تر از چیزی است که فکر می‌کردی.

کد تمیز خودش را توضیح می‌دهد.

اگر لازم شد کنار هر چند خط، جمله‌ای بنویسی تا بگویی «این قسمت فلان کار را می‌کند»، یعنی باید کد را بهتر بنویسی، نه کامنت بیشتر.

چگونه از کامنت‌ها درست استفاده کنیم

۱. قبل از نوشتن کامنت، بپرس «آیا می‌توانم این را در خود کد واضح‌تر کنم؟» گاهی فقط با تغییر نام تابع یا متغیر، به کامنت نیاز نخواهد بود.
۲. کامنت باید توضیح چرا باشد، نه چه. کد می‌گوید «چه کاری انجام می‌شود»، کامنت باید بگوید «چرا این‌طور انجام می‌شود».
۳. از کامنت برای هشدار، فرض، یا تصمیم خاص استفاده کن.
۴. کامنت نباید تکرار کد باشد. اگر فقط می‌گویی چه نوشته‌ای، داری نویز اضافه می‌کنی.
۵. کامنت‌ها را به‌روز نگه دار یا حذفشان کن.
- کامنت اشتباه از نبودن بدتر است. چون اعتماد خواننده را از بین می‌برد.
۶. از کد تمیزتر برای کاهش نیاز به کامنت استفاده کن. به جای نوشتن:

○ ○ ○

```
// کاربر را ذخیره و سپس ایمیل خوش‌آمد ارسال می‌کند  
saveUserAndSendEmail(user);
```

بهتر است دو تابع بنویسی:

○ ○ ○

```
saveUser(user);  
sendWelcomeEmail(user);
```

”کد بد را توضیح نده — بازنویسی‌اش کن.“ — رابرت سی. مارتین

اصل ۴: قالب‌بندی؛ صدای بی‌کلام کد

”کد تمیز، با نگاه خوانده می‌شود، نه با تلاش.“



خوانایی یعنی سرعت درک.

هر بار که دیگران برای فهم ساختار کدت مکث می‌کنند، از سرعت تیم کاسته می‌شود.

هر زبان، قواعد خودش را دارد، اما اصل مشترک این است: چشم باید راحت بخواند.

چگونه قالب‌بندی تمیز داشته باشیم

۱. کد را طوری بنویس که با نگاه، ساختارش دیده شود.
ساختارهای تو در تو را با تورفتگی مشخص کن تا چشم مسیر را گم نکند.
۲. بین بخش‌های مرتبط، فاصله بگذار؛ بین ایده‌های متفاوت، فضا بده.
فاصله عمودی همان کاری را می‌کند که نقطه و پاراگراف در متن.
کد فشرده‌تر، لزوماً بهتر نیست — فقط سخت‌تر است.
۳. خطوط مرتبط را کنار هم نگه دار.
اگر تغییری فقط در یک بخش استفاده می‌شود، همان‌جا تعریفش کن.
پراکندگی یعنی زحمت بی‌دلیل برای خواننده.
۴. تابع‌ها و ماژول‌ها را با نظم درونی بنویس.
در هر سطح از کد (کلاس، تابع، فایل)، ترتیب منطقی از بالا به پایین حفظ شود:
تعریف عمومی در بالا، جزئیات اجرایی در پایین.
۵. از چیدمان سازگار استفاده کن.
همه‌ی اعضای تیم باید یک سبک واحد داشته باشند.
فاصله‌ها، براکت‌ها و ترتیب آرگومان‌ها را مثل قرارداد ببین، نه سلیقه.
۶. طول خطوط را کنترل کن.
کدی که از صفحه بیرون می‌زند، ذهن را هم بیرون می‌برد.
هر خط باید با یک نگاه کامل دیده شود.
۷. خوانایی را فدای فشردگی نکن.
کدی که «زیبا و کوتاه» است اما فهمیده نمی‌شود، تمیز نیست.
در تمیزی، اولویت با فهم است، نه فشردگی.

اصل ۵: خطاها را مثل داده مدیریت کن



بیشتر کدها وقتی درست کار می‌کنند، ساده‌اند. اما وقتی خطا پیش می‌آید، چهره‌ی واقعی طراحی نمایان می‌شود.

کدی که با خطاهای احتمالی رفتار حساب‌شده‌ای ندارد، مثل هواپیمایی است که فقط برای پرواز در هوای صاف ساخته شده باشد.

چگونه خطا را تمیز مدیریت کنیم

۱. خطا را پیش‌بینی کن، نه سرکوب.

هر چیزی که ممکن است خطا دهد، باید جایی برای واکنش داشته باشد. با try-catch فقط مشکل را دفن نکن — فکر کن در صورت رخ دادن خطا، سیستم باید چه رفتاری نشان دهد.

۲. از استثنا (Exception) به‌جای کد بازگشتی استفاده کن.

برگرداندن مقادیر خاص مثل -1 یا null برای اعلام خطا، خوانایی را می‌کشد. استثنایا را به‌جا و با معنا پرتاب کن.

۳. پیام خطا باید شفاف و انسانی باشد. خطا باید به فهم کمک کند، نه به ترس.
به جای Error 400، بنویس:

User not authorized to access this resource.

۴. هر catch باید هدف مشخصی داشته باشد.
اگر فقط بنویسی:

○ ○ ○

```
catch (Exception e) {}
```

در واقع داری به سیستم می‌گویی: «هر چه شد، وانمود کن نشد».
این یعنی خودفریبی فنی.

۵. مسئولیت مدیریت خطا را در جای درست بگذار.

کدی که خطا را ایجاد می‌کند، لزوماً نباید آن را برطرف کند.
تابع پایین‌دستی فقط گزارش می‌دهد؛ تابع بالادستی تصمیم می‌گیرد چه کند.

۶. از بلاک‌های try کوچک و تمیز استفاده کن.

وقتی کل تابع را درون try می‌گذاری، در واقع داری چشم‌پسته رانندگی می‌کنی.
بخش مستعد خطا را جدا کن تا رفتار قابل پیش‌بینی بماند.

۷. داده‌های ناقص را زود شناسایی کن.

اگر ورودی‌ها را در ابتدای مسیر اعتبارسنجی کنی، نیمه دوم برنامه‌ات در آرامش خواهد بود.

۸. در تست‌ها، خطا را هم آزمایش کن.

کد تمیز فقط در حالت موفق تست نمی‌شود؛ باید بدترین سناریوها را هم
تاب بیاورد.

اصل ۶: داده و رفتار را از هم جدا نکن



کد تمیز فقط به این فکر نمی‌کند چه کاری انجام دهد، بلکه مهم است کجا باید آن کار انجام شود.

بیشتر کدهای نامرتب از یک اشتباه ساده شروع می‌شوند: نوشتن تابع‌هایی که روی داده‌هایی کار می‌کنند که به آن‌ها تعلق ندارند.

مثلاً:

○ ○ ○

```
function calculateSalary(employee) {  
    return employee.base + employee.bonus;  
}
```

اینجا تابع بیرونی دارد داده‌ی employee را باز می‌کند و با جزئیاتش کار می‌کند. یعنی منطق محاسبه‌ی حقوق، بیرون از خود کارمند نوشته شده — و هر جا داده‌ی employee تغییر کند، این تابع هم باید تغییر کند. **کد تمیز این گونه نیست.** در طراحی تمیز، داده و رفتار در یک مکان زندگی می‌کنند.

همان مثال، به صورت تمیزتر:



```
employee.calculateSalary();
```

حالا شیء employee خودش می‌داند چطور باید حقوقش را حساب کند. اگر بعدها ساختار حقوق تغییر کند، فقط درون همان کلاس اصلاح می‌شود، نه در ده‌ها فایل دیگر.

چند قانون ساده برای طراحی تمیزتر

- ۱. هر داده باید در جایی نگهداری شود که رفتار مرتبط با آن هم وجود دارد.** اگر شیء Invoice داری، متدهایی مثل `calculateTotal()` و `applyDiscount()` باید درون همان باشد، نه بیرون.
- ۲. شیء نباید فقط ظرف داده باشد.** اگر کلاسی فقط متغیر دارد و هیچ رفتاری ندارد، در واقع فقط یک ظرف داده است، نه یک شیء. شیء واقعی باید خودش بداند با داده‌هایش چه کند — نه این‌که منتظر باشد دیگران بیایند و آن را دستکاری کنند.
- ۳. از `getter/setter` فقط وقتی لازم است استفاده کن.** به جای اینکه اجازه دهی بقیه کدها مقدار فیلدها را بگیرند و تغییر دهند، تابع یا متدی بنویس که آن رفتار را خودش انجام دهد. کد تمیز با «داده در دسترس» تعریف نمی‌شود، با «رفتار کنترل‌شده» تعریف می‌شود.
- ۴. در مرز سیستم (مثل API یا انتقال داده) داده‌ها می‌توانند ساده باشند.** اما در منطق درونی برنامه، رفتار باید همراه داده بماند.

اصل ۷: مرزها را شفاف بساز، نه شکننده



هر برنامه‌ای از دو بخش ساخته می‌شود:

چیزی که خودت نوشته‌ای، و چیزهایی که دیگران ساخته‌اند و تو فقط ازشان استفاده می‌کنی — کتابخانه‌ها، سرویس‌ها، API‌ها و ابزارهای بیرونی.

وقتی این دو بخش از هم جدا نباشند، کد تو وابسته و شکننده می‌شود. یعنی هر بار بیرون تغییر کند، تو هم باید همراهش تغییر کنی. طراحی تمیز یعنی کشیدن یک خط روشن بین کد خودت و کد دیگران.

چگونه مرزهای تمیز بسازیم

۱. فقط یک نقطه تماس با دنیای بیرون داشته باش.

اگر از یک کتابخانه خارجی استفاده می‌کنی، دسترسی به آن باید در یک فایل یا ماژول مشخص انجام شود، نه در همه پروژه. مثلاً اگر همه‌جا مستقیماً از axios استفاده کنی، تغییر آن به fetch یا GraphQL به کابوس تبدیل می‌شود. ولی اگر همه درخواست‌ها فقط از مسیر api.js عبور کنند، با یک تغییر در آن فایل، کل سیستم به‌روزرسانی می‌شود.

۲. با Interface فکر کن، نه با Implementation.

به‌جای اینکه کد به جزئیات کتابخانه وابسته باشد، رفتار مورد نیاز را در قالب یک رابط (interface) تعریف کن.

به‌جای:

```
axios.get('/users')
```

بنویس:

```
httpClient.getUsers()
```

و بعد تصمیم بگیر که پشت httpClient چه کتابخانه‌ای کار کند.

۳. وابستگی را تزریق کن

تابعی که خودش شیء وابسته را می‌سازد، در آینده آزادی عمل ندارد. به‌جایش وابستگی را از بیرون بگیر تا بتوانی هر زمان آن را عوض کنی. (این همان Dependency Injection است، یعنی جداکردن تصمیم از استفاده.)

۴. برای وابستگی‌های بیرونی "لایه محافظ" بساز.

اگر با سرویس پرداخت، API کاربر، یا دیتابیس بیرونی کار می‌کنی، در میانه مسیر، یک لایه بنویس (مثلاً PaymentService یا UserGateway) تا کد اصلی تو مستقیم به بیرون وصل نباشد.

۵. با تست از مرزها محافظت کن.

برای لایه‌های مرزی، تست‌های جدا بنویس تا مطمئن شوی اگر رفتار بیرون عوض شد،

حداقل بفهمی از کجا آسیب می‌خوری.

وابستگی درهم:

○○○

```
import axios from "axios";  
const data = await axios.get("/products");
```

اگر بخواهی Axios را حذف یا جایگزین کنی، باید همه جا را بگردی.
مرز تمیز:

در این حالت، فقط ماژول api.js با Axios آشناست، بقیه‌ی سیستم فقط با مفهوم "api" حرف می‌زنند — نه با جزئیاتش.
در نهایت مرز تمیز یعنی استقلال.
وقتی بین سیستم خودت و دنیای بیرون دیواری شفاف داری، تغییر دیگران تو را نمی‌لرزاند.

«کد تمیز یعنی کنترلی که از دستت بیرون نرود.»

اصل ۸: تست نویسی



کدی که نمی‌شود برایش تست نوشت، یعنی آن قدر درهم است که خودش هم نمی‌فهمد چه می‌کند.

رابرت مارتین در این بخش می‌گوید: “ If it's not tested, it's broken. ”

یعنی اگر چیزی تست ندارد، باید فرض کنی خراب است.

اما نکته فقط «نوشتن تست» نیست؛ او درباره‌ی طراحی کدی که تست‌پذیر باشد حرف می‌زند — کدی که می‌شود بدون دردسر، بخش به بخش بررسی‌اش کرد، چون منطق‌ها جدا، مرزها تمیز و رفتارها قابل پیش‌بینی‌اند.

چگونه تست تمیز بنویسیم

۱. تست باید ساده‌تر از خود کد باشد.

تست قرار نیست خلاقانه یا پیچیده باشد. اگر تست‌نویسی تو نیاز به توضیح دارد، احتمالاً داری همان اشتباهات کد اصلی را تکرار می‌کنی.

۲. هر تست باید یک چیز را بررسی کند.

اگر در یک تست چند حالت و چند انتظار داری، در واقع چند تست نوشته‌ای در قالب یکی. تست تمیز دقیق است، نه پرحجم.

۳. نام تست باید بگوید چه رفتاری را می‌سنجد.

نام‌های مبهم مثل `test1` یا `mainCase` هیچ کمکی نمی‌کنند. به جایش بنویس:

`shouldReturnErrorWhenUserIsUnauthorized()`

خواننده باید بدون نگاه به کد، بفهمد هدف تست چیست.

۴. کد تست هم باید خوانا و منظم باشد.

تست هم کد است، و بی‌نظمی در آن، باعث بی‌اعتمادی به نتایجش می‌شود. ساختار Arrange-Act-Assert را رعایت کن:

○ آماده‌سازی داده

○ اجرای عمل

○ بررسی نتیجه

۵. از تست‌های شکننده پرهیز کن.

تستی که با هر تغییر کوچک در کد می‌شکند، مفید نیست. تست باید منطق را بررسی کند، نه جزئیات پیاده‌سازی.

۶. همه چیز را تست نکن؛ چیزهای مهم را تست کن.

تست خوب مثل نورافکن است، نه چراغانی. روی رفتارهای حیاتی تمرکز کن، نه هر خط از کد.

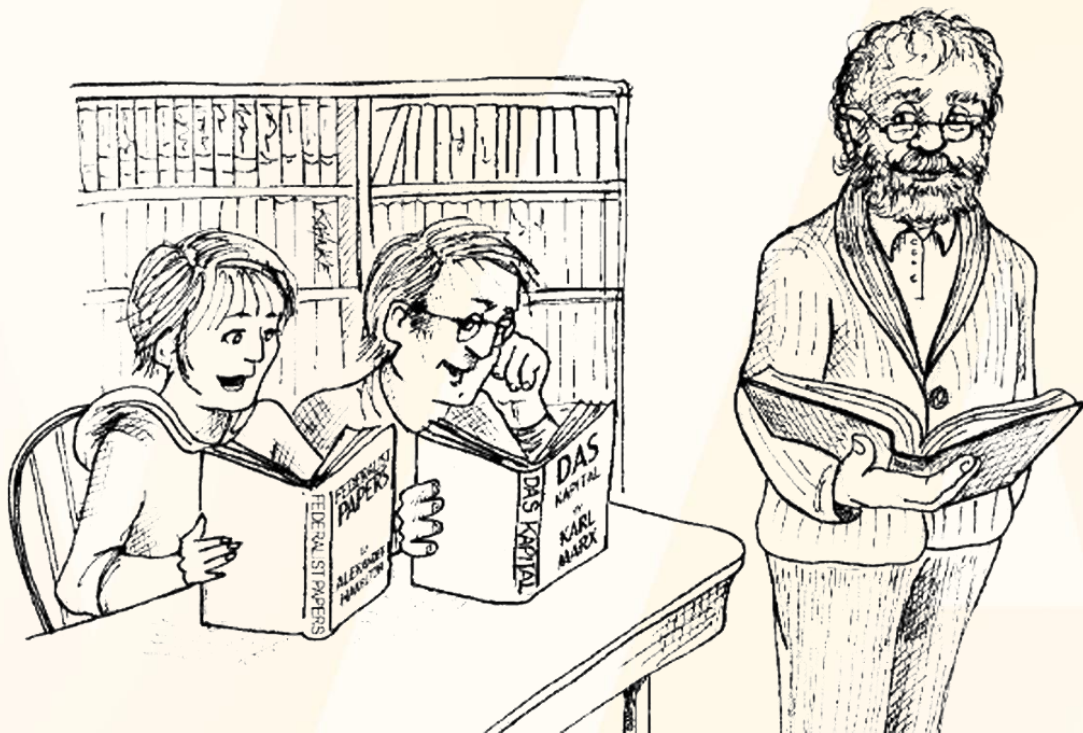
۷. تست باید سریع و مستقل باشد.

تستی که به دیتابیس یا سرویس بیرونی وابسته است، باید شبیه سازی (Mock) شود. هر تست باید بتواند به تنهایی اجرا شود.

۸. تست های قدیمی را نگه دار و به روز کن.

تست قدیمی مثل بیمه نامه است — تا وقتی درست نگهداری شود، ارزش دارد.

اصل ۹: کلاس تمیز یعنی نقش روشن و وابستگی کم



رابرت مارتین در این فصل می‌گوید:

” کلاس خوب فقط مجموعه‌ای از تابع‌ها نیست؛ بلکه قراردادی روشن برای رفتار، تغییر و رشد سیستم است.“

کد تمیز از اینجا به بعد دیگر فقط درباره‌ی نحو و نام‌گذاری نیست، بلکه درباره‌ی «ساختار فکر» است.

کلاس تمیز مثل تیم منظم است: هر عضو فقط کار خودش را می‌داند، اما با بقیه هماهنگ عمل می‌کند.

کلاسی که از بیرون همه‌چیز را کنترل می‌کند، یا از درون به همه‌چیز وابسته است، نه تمیز است، نه قابل اعتماد.

چگونه کلاس تمیز طراحی کنیم

۱. هر کلاس فقط باید یک مسئولیت داشته باشد.

یعنی فقط یک دلیل برای تغییر. اگر کلاسی هم داده را مدیریت می‌کند، هم منطق را، هم ذخیره‌سازی را، در واقع سه کلاس در یک پوشش ساخته‌ای. ☒ درست:

UserRepository برای ذخیره و بازیابی کاربر
UserService برای منطق و قوانین کاربر
UserController برای دریافت و پاسخ درخواست‌ها
هرکدام فقط کار خودشان را انجام می‌دهند.

۲. وابستگی‌ها را به حداقل برسان.

کلاسی که به ده کلاس دیگر نیاز دارد تا کار کند، مثل دستگاهی است که با ده کابل به برق وصل است — کوچک‌ترین تغییر، همه‌چیز را خاموش می‌کند. وابستگی‌ها را با تزریق (Dependency Injection) یا Interface مدیریت کن، نه با ساخت مستقیم اشیا درون کلاس.

۳. انسجام (Cohesion) را حفظ کن.

تمام متدهای یک کلاس باید درباره‌ی «یک مفهوم واحد» حرف بزنند. اگر دو متد از داده‌های کاملاً متفاوت استفاده می‌کنند، احتمالاً در کلاس اشتباهی قرار گرفته‌اند. کلاسی با انسجام بالا یعنی کلاسی که می‌توانی آن را خلاصه کنی در یک جمله:

«این کلاس کار X را انجام می‌دهد.»
اگر برای توضیحش باید از "و" یا "هم" استفاده کنی، انسجام نداری.

۴. داده‌های درونی را پنهان کن.

متغیرهای عضو باید خصوصی باشند. کلاس نباید داده‌اش را در اختیار دیگران بگذارد؛ بلکه باید از طریق رفتار (متد) با بیرون ارتباط بگیرد.

۵. کلاس را کوچک نگه دار.

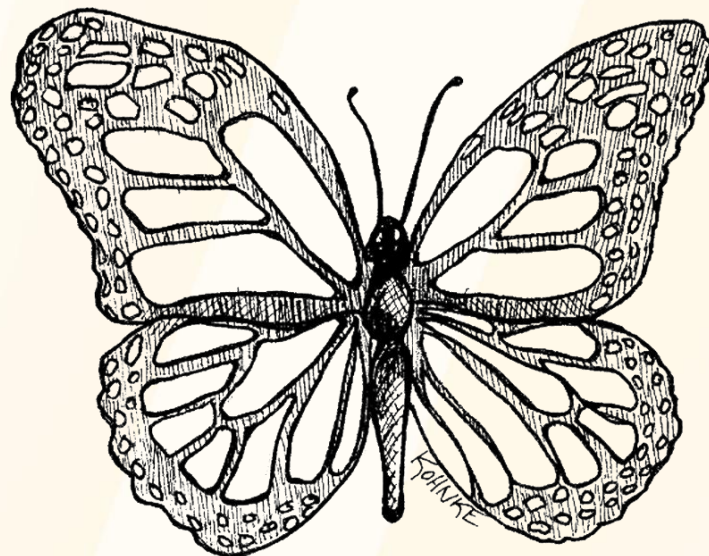
کوچکی یعنی تمرکز. وقتی فقط یک نقش داری، طبیعی است که مختصر بمانی.

۶. ارتباط بین کلاس‌ها باید از طریق قرارداد باشد.

یعنی هر کلاس فقط بداند دیگری چه کاری انجام می‌دهد، نه چطور انجامش می‌دهد.

وقتی کلاس‌ها به رفتار هم متکی‌اند، نه به جزئیات هم، کدت قابل تغییر و جایگزینی می‌شود — بدون شکستن زنجیره.

اصل ۱۰: طراحی تمیز، ساخته نمی‌شود — رشد می‌کند



اینجا رابرت مارتین دیگر درباره‌ی تابع یا کلاس حرف نمی‌زند، بلکه درباره‌ی «چگونه تمیزی را در طول زمان حفظ کنیم» صحبت می‌کند. می‌گویند طراحی خوب، از قبل کامل نیست؛ در اثر نوشتن، تست کردن و بازنویسی مداوم رشد می‌کند — درست مثل ذهن برنامه‌نویس حرفه‌ای.

چهار ستون طراحی تدریجی

۱. همه تست‌ها باید سبز باشند.
تست‌ها چراغ‌های مسیرند — اگر حتی یکی خاموش باشد، مسیر امن نیست.
۲. هیچ تکراری در کد نباید بماند.
هر تکرار یعنی هزینه‌ی آینده. وقتی دو بخش از کد یک کار مشابه انجام می‌دهند، یعنی فرصتی برای استخراج مفهوم مشترک داری.
۳. نیت کد باید واضح باشد.
هر تابع و کلاس باید هدف خود را واضح بیان کند. اگر لازم است توضیحش دهی، هنوز تمیز نشده.
۴. کمترین تعداد کلاس و متد ممکن را داشته باش.
سادگی یعنی قدرت. هر چیزی که ارزش واقعی اضافه نمی‌کند، باید حذف شود. سیستم خوب، از بیرون ساده است حتی اگر درونش پیچیده باشد.

نکات پایانی: تمیزی در عمل

در دنیای واقعی، کد تمیز مثل تناسب اندام است: یک بار که به‌دست آمد، باید نگه داشته شود — هر روز، با انضباط.

۱. بازنویسی دشمن سرعت نیست، متحدِ دوام است
بسیاری از تیم‌ها از بازنویسی می‌ترسند چون فکر می‌کنند «وقت تلف می‌کند.» اما مارتین می‌گوید:
«هر بار که بازنویسی نمی‌کنی، در واقع در حال ساخت بدهی آینده‌ای»

بازنویسی یعنی اصلاح طراحی بر اساس چیزهایی که امروز می‌دانی، نه اصرار بر تصمیماتی که دیروز با اطلاعات کمتر گرفته‌ای.

در عمل: هر بار که فیچری تمام شد، ۵ دقیقه وقت بگذار و از خودت بپرس: آیا این کد اگر دو ماه دیگر بخواهد تغییر کند، واضح است؟

۲. تمیزی کار تیمی است، نه مهارت فردی

کد تمیز فقط زمانی ماندگار است که همه‌ی اعضای تیم به آن متعهد باشند. یک نفر تمیزکار در میان ده نفر بی‌نظم، مثل جاروکردن خانه‌ای است که بقیه هنوز دارند درش بتن می‌ریزند.

در عمل: استانداردهای تیمی بنویس — از نام‌گذاری تا ساختار فایل و نحوه‌ی تست.

همه باید از یک زبان تمیزی استفاده کنند.

۳. کامنت، کامپایلر ذهن است

در پروژه‌های واقعی، مارتین بارها نشان می‌دهد که به‌جای نوشتن کامنت برای توضیح، کد را بازطراحی می‌کند تا توضیح بی‌نیاز شود.

در عمل: هر جا می‌خواهی کامنت بنویسی، اول بپرس:

«آیا می‌توانم این توضیح را با نام بهتر یا ساختار ساده‌تر جایگزین کنم؟» اگر بله، یعنی هنوز فرصت تمیز کردن داری.

۴. تست‌ها فقط برای اطمینان نیستند — ابزار طراحی‌اند

در پروژه‌های واقعی کتاب، نویسنده قبل از نوشتن منطق اصلی، تست را می‌نویسد تا مطمئن شود درک درستی از رفتار دارد.

این دقیقاً فلسفه‌ی (Test-Driven Development) TDD است: نوشتن کد بر اساس رفتار مورد انتظار، نه بر اساس حدس.

در عمل: قبل از نوشتن تابع، سه سؤال بپرس:

○ این تابع دقیقاً چه ورودی می‌گیرد؟

○ چه خروجی می‌دهد؟

○ در صورت خطا، باید چه کند؟

پاسخ این سه سؤال همان تست تست‌توست.

۵. تمیزی یعنی حذف، نه افزودن

در پایان کتاب، مارتین می‌گوید:

”برنامه‌نویس حرفه‌ای، کسی است که می‌داند چه چیزی را ننویسد.“

سیستم تمیز از انباشت درست نمی‌شود؛ از حذف‌های درست ساخته می‌شود.
در عمل: وقتی تردید داری یک بخش از کد را نگه داری یا نه، حذفش کن —
اگر واقعاً لازم باشد، به شکلی تمیزتر برمی‌گردد.

درنهایت، کد تمیز یک سبک نیست، یک منش است. یعنی باور به اینکه هر خط کد،
مسئولیتی دارد:

**شفاف بودن، ساده بودن، و احترام گذاشتن به خواننده‌ی بعدی — حتی اگر آن
خواننده، خودت باشی.**

”برنامه‌نویسی تمیز یعنی صداقت با آینده“

Robert C. Martin —



سون لرن

دوره‌های آموزشی سون لرن، جایی برای کسب تجربه و مهارت از بهترین مدرسین برنامه‌نویسی ایران

مشاوره رایگان

جزئیات بیشتر