PDF

آکادمی متنو
Learn.matno.co

کدنویسی تمیز چیست؟

راهکارهای نوشتن کد خوانا

نویسنده: محمدحسین میثاق‌پور

بسم الله الرحمن الرحيم

فهرست مطالب

بخش صفرم – مقدمه

بخش یکم – چیه و چرا؟

بخش دوم – نظریه اساسی خوانایی کد

بخش سوم – نام‌گذاری

بخش چهارم – زیبایی شناسی

بخش پنجم – کامنت‌گذاری

بخش ششم – ساده‌سازی جریان برنامه

بخش صفرم

مقدمه

کدنویسی تمیز (Clean Code) اصطلاحی هستش که برای اولین بار توسط آقای رابرت سی مارتین که به عمو باب هم معروفه ابداع شد. ایشون طی تجربیاتی که از کار کردن با کدها داشته، متوجه اهمیت خوانا بودن کدها میشه و تصمیم می‌گیره تا دانسته‌های خودش رو با بقیه هم اشتراک بذاره. ایشون چندین کتاب در زمینه‌ی مهندسی نرم‌افزار منتشر کرده‌اند. که یکی از محبوب‌ترین اونها کتاب Clean Code هستش. در این کتابچه قصد داریم تا با مفهوم کدنویسی تمیز بیشتر آشنا بشیم و در ادامه یه سری راهکارها برای خوانا شدن کدها ارائه کنیم.

احتمالا برای شما هم زمان‌هایی پیش اومده که در حال بررسی کدهای یک پروژه جدید بودین و احساس سردرگمی کردین! مدت زیادی رو صرف بررسی و مطالعه کدها کردین اما یه حس بدی داشتین! مثلا این که منطق کدها به چه صورت هستش و کلا یه احساس سردرگمی.

یا مثلا بعد از مدت‌ها، تصمیم گرفتین تا به برنامه‌ای که قبلا خودتون نوشتید، مراجعه کنید اما این بار نیز همون احساس مشابه رو داشتین و از خودتون می‌پرسید این کدها چه جوری کار می‌کنه؟ من تا چند ماه پیش که به این مطالب مسلط بودم اما الان نمی‌تونم درک‌اش کنم؟

احتمالا در این مواقع شما با کدهای کثیف روبرو شدین. حالا می‌پرسید کدهای کثیف چی هستند؟ این موضوعی هستش که در این کتابچه قصد داریم به اون بپردازیم.

برای اینکه این موضـوع رو روشـن‌تر کنم بیاین با یه مثال بررسـی اش کنید. دو اتاق رو در نظر بگیرین. یه اتاق مرتب و تمیز و دیگری اتاق کثیف و نامرتب. در دنیای برنامه‌نویسی میشه گفت کدنویسـی تمیز یه جورایی مثل یک اتاق مرتب و منظم می‌مونه و در مقابل کدکثیف مثل اون اتاق نامرتب هستش.

همونطور که وقتی ما وارد یک اتاق مرتب میشـیم و حس و حال خوبی داریم همونطور هم وقتی سراغ کدهای تمیز میریم احساس خوبی داریم. توی یه اتاق تمیز به راحتی میشه به اشیا دسـترسـی داشـت. اگه لازمه بعضـی از جاهاش رو تغییر میدیم و در مجموع یه حس مطلوبی داریم و به طور مشابه چنین ویژگی‌هایی در کد خوانا وجود داره.

اما در مقابل یه اتاق کثیف و نامرتب رو تصـور کنید. مثلا موقع راه رفتن تو اون اتاق ممکنه یه شـی تیزی تو پاتون فرو بره. تصـور کنید قراره یه کتاب از قفسـه بردارین احتمال اینکه یه چیزی بیفته تو سـرتون وجود داره و در مجموع این اتاق نامرتب حس خوبی به ما نمیده و به طور مشابه کدهای کثیف چینن ویژگی‌هایی دارند.

درسته هر دو شون یه شباهتی دارند و اونم اینه که به نحوی وسایل مورد نیاز داخل اش به یه ترتیبی قرار گرفته اند اما تفاوت شون حسی هستش که از اونا می‌گیریم.

فکر می‌کنم تا به اینجای کار با مفهوم کدنویسـی تمیز به طور نسـبی آشـنا شـدین. توی این کتابچه قراره به مباحث زیر پرداخته بشه.

کدنویسی تمیز چیست؟

ویژگی‌های کد تمیز چیست؟

چرا نوشتن کدهای خوانا مهم است؟

دلایل ناخوانا شدن کدها چیست؟

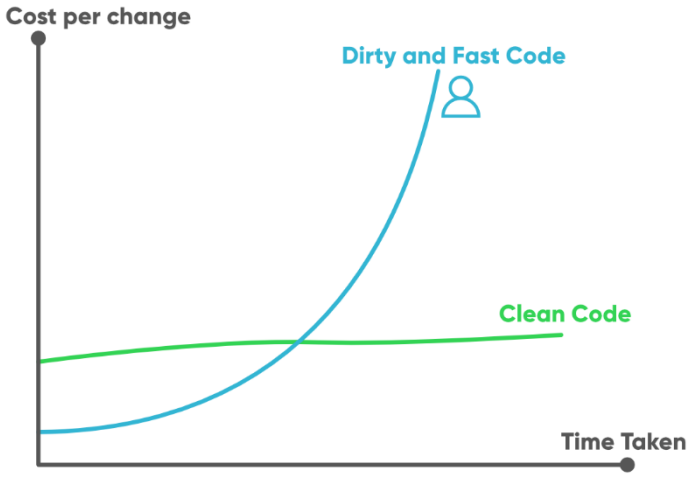
و راهکارهایی برای نوشتن کد خوانا

بخش اول

چیه و چرا؟

چرا کدنویسی تمیز مهمه؟

کدنویسی تمیز اهمیت زیادی در توسعه و نگه‌داری نرم‌افزار داره. اجازه بدین که با یک نمودار اهمیت این مساله رو واضح‌تر کنم.



همونطور که در نمودار می‌بینید، منحنی آبی رنگ مربوط به نرخ رشد هزینه کدنویسی ناخوانا و منحنی سبز رنگ مرتبط با یه کد تمیز هستش.

در شروع زمان توسعه اگرچه کدنویسی کثیف هزینه‌ی کمتری رو از ما می‌گیره و در مقابل کدنویسی خوانا هزینه‌ی بیشتری صرف می‌کنه، اما در ادامه میزان هزینه بر اساس تغییرات در منحنی کد کثیف شتاب بیشتری می‌گیره و با گذشت زمان این مقدار به طور صعودی بیشتر میشه.

کدنویسی تمیز اگرچه هزینه‌ی ابتدایی اون زیاد هستش و علت اون هم برنامه‌ریزی و بررسی برای پیاده‌سازی یک ساختار منظم هستش، اما تقریبا نرخ رشدش ثابت هستش و با گذر زمان میزان هزینه‌های پروژه تغییر چندانی نمی‌کنه.

و اینجا هستش که اهمیت کدنویسی تمیز مشخص میشه.

چون اون چیزی که در برنامه‌نویسی اهمیت داره داره میزان توسعه پذیری اون هستش و اینکه پروژه ما در آینده قابلیت اعمال تغییرات رو داشته باشه.

کدنویسی تمیز چه مزایایی داره؟

در ادامه قصد دارم تا به بررسی سه مورد از مزایای کدنویسی خوانا بپردازم.

آسان‌بودن برای خواندن

در مقدماتی‌ترین سطح، اون شخصی که داره کدهای ما رو می‌خوانه باید به راحتی بتونه متوجه منطق و روال کدها بشه و خوندن براش آسون بشه. از ویژگی‌های کدهای تمیز، واضح بودن و ابهامات کم در اون هستش.

آسان بودن برای تغییر

یکی دیگه از ویژگی‌های کدهای تمیز، آسان بودن برای تغییر هستش. یکی از بخش‌های جدایی ناپذیر فرآیند توسعه نرم‌افزار، اعمال تغییرات هستش. کدهای تمیز به گونه‌ای نوشته میشن که خواننده به راحتی و با کمترین هزینه قادره که در صورت نیاز تغییراتی رو اعمال کنه بدون این که بخش‌های دیگه آسیب پذیر بشند.

آسان بودن برای رفع عیب

از ویژگی‌های دیگه کدهای خوانا، قابلیت رفع عیب آسان اونا هستش. در فرآیند توسعه نرم‌افزار گاهی اوقات مشکلاتی و عیب‌هایی به وجود میاد. در قدم اول برنامه‌نویس باید قادر

به کشف اون باگ‌ها بشه و در مرحله بعد بتونه اونا رو تغییر بده. کدهای خوانا قابلیت پیمایش و دسترسی خوبی به بخش‌های مختلف دارند و این امکان رو به ما میدن که سریع‌تر باگ‌ها رو کشف و رفع شون کنیم.

چرا کدهای ما کثیف میشه؟

خب در این بخش قصد داریم به این موضوع بپردازیم که چرا کدهای ما کثیف میشن. در ادامه به بررسی دو علت عمده در نامرتب شدن کدها پرداخته میشه.

عدم تجربه‌ی کافی در برنامه‌نویسی

یکی از دلایلی که باعث ناخوانا شدن کدها میشه، عدم تجربه‌ی کافی برنامه‌نویس هستش. مثلا فردی رو تصور کنید که به تازگی برنامه‌نویسی رو یاد گرفته و با سینتکس یه زبان آشنا شده، طبعیتا این فرد در ابتدای مسیر خودش با روش‌های کدنویسی تمیز آشنا نیست. و تمرکزش بر اینه که یه کدی بنویسه که خروجی صحیحی داشته باشه. برای مثال ممکنه که کل برنامه‌ی خودش رو داخل یه تابع طولانی تعریف کنه یا مثلا از نام‌گذاری‌های مناسبی استفاده نکنه. که در مجموع این موارد باعث ناخوانا شدن کد میشن.

عجله در تحویل پروژه

عامل بعدی در نامرتب شدن کدها، عجله در تحویل پروژه هستش. گاهی اوقات برنامه‌نویس به واسطه‌ی اجبار کارفرما مجبور میشه که پروژه رو توی یه بازه‌ی زمانی خیلی فشرده‌ای تحویل بده. در چنین شرایطی اگه حتی اون برنامه‌نویس یه برنامه‌نویس نسبتا حرفه‌ای باشه و با اصول کدنویسی تمیز آشنا باشه ممکنه که اصول تمیزنویسی کد رو فدای زمان بکنه و صرفا یه کدی رو بنویسه و پروژه رو تحویل بده و بره. اما چه بسا که چنین کاری در آینده مشکلات و یا اصطلاحا بدهی‌های فنی (Technical Debt) به وجود بیاره.

بخش دوم

نظریه اساسی خوانایی کدها

در این قسمت قراره به نوعی طرز نگرش در کدنویسی بپردازیم.

برخی از برنامه‌نویسان فکر می‌کنند که جزء انسان‌های باهوش و خفن هستند و همین موضوع ممکنه در کدنویسی اونا هم اثر بذاره و مثلا کدهاشون رو رمز آلود بنویسند. به خیال خودشون ممکنه کدهای خفنی نوشته باشند اما از طرفی آیا این کدهایی که نوشتند رو بقیه هم می‌تونند متوجه بشند؟ ممکنه طرف یه برنامه‌نویس فریلنسر باشه و به صورت تیمی کار نکنه و انفرادی باشه و استدلال‌اش اینه که کدهای خودش رو می‌تونه متوجه بشه. اما یه سوال... اگه بعد از گذشت چند ماه اون برنامه‌نویس دوباره سراغ کدهای خودش بیاد آیا متوجه منطق اون میشه؟ یا اینکه مجبوره دوباره از اول زمان بذاره و کدها رو متوجه بشه؟

ایده ی کلیدی و اساسی: آسان بودن کد

در اینجا قصد دارم که مهم‌ترین اصل و ایده‌ی اساسی در کدنویسی رو بگم و اون هم آسان نوشتن کدها هستش. شما به عنوان یه برنامه‌نویس باید تا اونجایی که میشه کدهاتون رو ساده‌تر بنویسید و تا اونجایی که میشه از پیچیدگی‌ها دوری کنین. جوری کدهاتون رو بنویسید که حتی یه آدم با آی کیو پایین هم بتونه کدهاتون رو متوجه بشه 🙂

در واقع کل چیزی که می‌خوایم در زمینه‌ی کدنویسی تمیز بگیم همین ساده نویسی هستش و در بخش‌های بعد قصد داریم به صورت جزئی‌تر این موضوع رو مورد بررسی قرار بدیم و اونا رو با مثال‌های بیشتری توضیح بدیم.

بررسی یک مثال

یه سوال! آیا کدنویسی تمیز همیشه به معنای کوتاه نویسی هستش؟ یعنی مثلا فرض کنید دو تا برنامه دارین که خروجی یکسانی دارند اما یکی از اونها شامل ۲۰ خط و دیگری شامل ۳۰ خط هستش. آیا لزوما یه کد خوانا و تمیز یه کد با حجم کم‌تر هستش؟ برای اینکه این موضوع روشن‌تر بشه بیاید مثال زیر رو بررسی کنیم.

```
assert((!(bucket = FindBucket(key))) || !bucket->IsOccupied());
```

در ادامه یه نمونه کد دیگه اورده شده که به نوعی مثال قبلی رو بازنویسی کرده.

```
bucket = FindBucket(key);
if (bucket != NULL) assert(!bucket->IsOccupied());
```

به نظر شما کدوم یکی از این دو مثال بالا خواناتر هستند؟

با کمی بررسی می‌تونید متوجه بشین که مثال دوم خواناتر هستش. اگرچه مثال دوم تعداد خط‌های بیشتری داره اما زمان مطالعه‌ی اون نسبت به مثال اول کم‌تر هستش. پس ملاک خوانایی تنها حجم کد نیست!

به عبارتی می‌تونیم بگیم مهم‌ترین اصل اساسی در کدنویسی تمیز کمینه‌کردن زمان مطالعه و بررسی کدهاست. یعنی هر چه قدر کدهای شما زمان کمتری رو برای مطالعه و خوندن بگیرن خواناتر هستند و لزوما کدهای با حجم کم خوانا نیستند. چون ممکنه حجم یه کد نسبتا کم باشه اما از طرف دیگه‌ای زمان بیشتری برای درک اون صرف بشه.

نظریه اساسی کد خوانا

کد باید به نحوی نوشته شود که زمان خواندن برای درک آن را به حداقل برساند.

به بیان دیگه، در کدنویسی تمیز، کوتاه و فشرده نوشتن مهم نیست، بلکه سریع متوجه شدن کد مهم هستش.

بخش سوم

نام‌گذاری

در این قسمت قصد داریم تا یه سری نکات برای نام‌گذاری‌ها رو با هم دیگه بررسی کنیم.

یکی از موارد تاثیرگذار در خوانایی کدها، انتخاب نام‌های صحیح هستش. قبل از اینکه وارد بحث نام‌گذاری در کدنویسی بشیم، بیاین به صورت کلی‌تر فلسله نام‌گذاری رو برررسی کنیم.

ما در دنیای واقعی برای اینکه بتونیم با هر آنچیزی که در پیرامون ما هستش از اشیاء گرفته تا اشخاص، دسترسی داشته باشیم، روی آنها نام‌گذاری می‌کنیم.

یه سری نکات وجود داره از جمله اینکه:

هر چه قدر نام‌های ما یکتاتر باشند اسامی ما خاص‌تر میشند و از حالت عمومی بودن خارج میشند.

هر چقدر نام‌های ما دقیق‌تر باشند و ابهام نداشته باشند باعث شفافیت بهتر میشند، مثلا توی دنیای واقعی یه سری کلمات چند پهلو هستند و دارای چند معنی هستند مثل کلمه شیر. آیا منظور از شیر، حیوان درنده شیر، یا شیر آب یا شیر گاو هستش؟

نام‌ها بسته به موقعیت مکانی که قرار دارند دسترس پذیر میشند. مثلا توی یه خانواده اسم یه نفر علی هستش، وقتی یکی از اعضای خانواده نام علی رو صدا می کنه مشخصه که منظور کیه. اما توی یه فضای بزرگتری مثل مدرسه، اگه معلم اسم علی رو صدا کنه اینجا ممکنه چندین شخص با نام علی وجود داشته باشند. در اینجا هستش که برای دقیق‌تر شدن از نام خانوادگی شخص استفاده میشه. به عبارتی در بعضی جاها برای اینکه نام‌ها دقیق‌تر بشند مجبوریم از توضیحات بیشتری استفاده کنیم.

و چندین نکته‌ی دیگه برای نام‌گذاری وجود داره که توضیح دادن اونا در این پست نمی‌گنجه. حالا که این موارد رو گفتم بیاین در دنیای برنامه‌نویسی، نام‌گذاری و اهمیت اونا رو بررسی کنیم.

نام‌ها، همان کامنت‌های کوچک هستند

اگه بخوایم یه تعریف جالبی از نام‌گذاری در برنامه‌نویسی داشته باشیم، بهتره که بگیم نام‌ها در واقع یه سری کامنت‌های خلاصه شده هستند. برای اینکه این موضوع روشن‌تر بشه بیان دو تا مثال زیر رو بررسی کنیم.

```
// 1
int d; // day
// 2
int day;
```

در مثال بالا یک متغیر با دو نام تعریف شده. یکی با نام d به همراه یک کامنت و دیگری با نام day و بدون کامنت.

به نظر شما کدوم یک از این دو متغیر خواناتر هستند؟ بله پاسخ day هستش. چون در اینجا دیگه نیازه به کامنت نیست.

به عبارت دیگه شما باید سعی کنید که نام‌گذاری‌هاتون به گونه‌ای باشه که نیازی به کامنت نباشه و هرآنچیزی که نیازه در خود نام قرار داده بشه.

در نام‌گذاری‌ها دقیق باشید

همیشه سعی کنید که نام‌های دقیقی برای متغیرها و سایر موارد دیگه انتخاب کنید و از اسامی عمومی حتی الامکان پرهیز کنید. مثلا کلمه get یه کلمه خیلی عمومی هستش و

دقیق نیست. مثلا توی برنامه آیا منظور از get ، دانلود (Download) یک فایل هستش؟ یا fetch کردن یه سری اطلاعات از پایگاه داده یا موارد دیگه؟

از دیکشنری استفاده کنید

به عنوان یه برنامه‌نویس، همیشه سعی کنید که برای انتخاب کردن نام‌های دقیق در برنامه‌تون از دیکشنری استفاده کنید. یه نمونه از دیکشنری‌های رایج google translate هستش.

از کلمات فینگلیش حتی الامکان پرهیز کنید

در برخی از کدهای افراد آماتور و تازه‌کار دیده شده که در نام‌گذاری‌ها از کلمات فینگلیش استفاده شده. مثلا برای تعریف یه متغیر به عنوان اعداد اول، از همچنین چیزی استفاده شده:

```
List<Int> adadAval = [];
```

تعریف چنین متغیرهایی ممکنه برای اون شخص و یا حتی توسعه‌دهندگان ایرانی دیگه قابل فهم باشه. اما یه لحظه از خودتون سوال کنید اگه یه توسعه‌دهنده خارجی زبان دیگه به این خط از برنامه برسه آیا متوجه این بخش میشه؟ بهتره که به جای تعریف این متغیر از متغیری با نام primeNumbers استفاده کنیم.

پس توصیه میشه که به عنوان یه برنامه‌نویس، همواره از استانداردها و اصول رایج در میان برنامه‌نویسان استفاده کنیم و اصطلاحا، موارد من در آوردی خودمون رو وارد این حوزه نکنیم 🙂

نام‌های ما چه قدر طولانی باشه؟

ببینید برای این مورد نمیشه یه اصل ثابتی رو تعریف کرد، اما اگه بخوایم یه معیاری رو برای میزان اندازه نام‌ها معرفی کنیم می تونیم بگیم:

اندازه نام هر متغیر متناسب با میزان اسکوپ دسترسی پذیری آن تغییر می‌کند.

حالا اینو که گفتم یعنی چه؟

توی برنامه‌نویسی یه بحثی هست به نام قابل دسترسی بودن یا متغیرهای محلی و سراسری .

بسته به محل و نوع تعریف متغیرها، بخش‌های دیگه از برنامه می‌توننـد به اون متغیر دسترسی داشته باشند. هر چه قدر تعداد خط های بیشتری از برنامه به اون متغیر دسترسی داشته باشند یا به عبارتی اون متغیر گلوبال تر باشه نام اون هم باید با جزییات بیشتری باشه. اگه دقت کرده باشین در ابتدای این پست مثال نام‌گذاری اشخاص رو زدم. مثلا توی یه مکانی مثل خانواده اشخاص کافیه که تنها اسم کوچک همدیگه رو صدا بزنند مثلا بگن علی. اما توی یه فضای بزرگتر و عمومی تر برای اینکه ابهام پیش نیاد اون شخص رو با نام خانوادگی اش صدا می زنند مثل علی احمدی.

نام‌های طولانی دیگر برای IDE ها مساله‌ای نیست

شاید یه عده‌ای به دلیل ترس از رخ دادن اشتباه تایپی در نام‌های طولانی، از تعریف نام‌های طولانی اجتناب کنند. اما امروزه با گسترش یافتن IDE های متنوع مثل vs code ، اندروید استودیو و ... به دلیل وجود قابلیت تکمیل خودکار یا Auto Completion این کار خیلی آسون شده. فقط کافیه که چند کاراکتر اول اون متغیری که تعریف کردین رو تایپ کنید تا IDEاز لیست پیشنهاداتش اونو براتون تکمیل کنه.

در قالب و فرمت نام‌گذاری‌ها ثبات داشته باشید

به طور کلی در نام‌گذاری متغیرها و موارد دیگه یه سری قالب‌هایی وجود دارند مثل **camelCase** یا **snake_case** و موارد دیگه.

همواره سعی کنید که از یه قالب ثابتی استفاده کنید. مثلا اگه در کل برنامه برای نام‌گذاری متغیرها از روش **camelCase** استفاده شده، نباید برای تعریف متغیر جدید از روش snake_case استفاده کرد. ثابت بودن نوع نام‌گذاری متناسب با نوع هر مولفه، باعث به وجود آمدن پیوستگی و انسجام میشه و خوانایی برنامه رو افزایش میده.

دقیق‌تر کردن نام‌گذاری‌ها با افزودن اطلاعات جزیی‌تر

همواره در انتخاب نام‌ها از خودتون این سوال رو بپرسین که آیا در آینده برای خودم و یا سایر کدنویس‌های دیگه این متغیر باعث به وجود اومدن ابهام و سوال نمیشه. اگه جواب مثبت هستش، سعی کنید با افزودن اطلاعات اضافه‌تر دیگه نام خودتون رو شفاف‌تر کنید. برای مثال:

```
int day;
```

منظور از متغیر day چی هستش؟ آیا منظور روز در هفته هستش؟ آیا روز در ماه هستش یا روز در سال؟

```
// which one do you mean?


int dayOfWeak;
int dayOfMonth;
int dayOfYear;
```

بخش چهارم

زیبایی شناسی

در این قسمت، قصد داریم تا به موضوع زیبایی‌شناسی در کدنویسی بپردازیم.

برای اینکه با اهمیت زیبایی شناسی در کدنویسی آشنا بشین بیان قبلش با یه مثال کار کنیم. احتمالا شما هم روزنامه‌ها و مجلات مختلفی رو دیدین که ظاهر گرافیکی خاصی دارند. و یه سری اصول ثابتی در اونها رایج هستش.

مثلا همه‌ی اونا دارای فهرست مطالب هستند که با ذکر کردن شماره هر صفحه متناسب هر موضوع، دسترسی به بخش‌های مورد نظر آسون‌تر میشه. عناصر و بخش‌های مختلفی مجله شامل متون و تصاویر با چینش و ترتیب و با ترازبندی خاصی کنار هم قرار گرفته‌اند و مثلا به صورت پراکنده نیستند. این کار نیز یکپارچگی بصری به مخاطب میده و خواننده احساس گیج‌بودن نمی‌کنه .یا مثلا فاصله‌ها و فضای خالی در میان عناصر از نسبت‌های خاصی پیروی می‌کنند.

و به طور کلی وقتی صحبت از زیبایی‌شناسی یه مجله میشه به ظاهر و عناصر بصری که شامل نحوه‌ی چینش عناصر، ترازبندی و ... هستش پرداخته میشه و با محتوای اون کار نداریم. به طور مشابه می‌تونیم چنین اصول زیبایی‌شناسی رو در کدنویسی تمیز هم داشته باشیم و با دانستن برخی نکات، یکپارچگی و انسجام متناسبی به کدهامون بدیم به نحوی که خواننده‌ی کدها، احساس خوبی از اون داشته باشه و کمتر سردگرم بشه.

حتی الامکان از IDE های رایج استفاده کنید

حتی الامکان سعی کنید از ویرایشگرهای متنی محبوب مثل Atom ، vs code و... استفاده کنید. چرا که با استفاده از IDE ها سرعت توسعه‌ی برنامه‌هامون سریع‌تر میشه و خیلی از کارها رو به صورت اتوماتیک برای ما انجام میده. یکی از قابلیت‌های رایجی که در IDE ها وجود

داره، منظم کردن فاصله‌ی کدها هستش. این فاصله‌ها هم شامل فاصله‌های عمودی بین خطوط و هم فاصله‌های افقی هستش. به عبارتی با این قابلیت نوعی ترازبندی در سطح کدها ایجاد میشه که باعث به وجود آمدن یکپارچگی بصری میشه و احساس خوبی رو به خواننده هنگامی بررسی کدها میده.

شکستن به فایل‌های جداگانه

یکی از مواردی که باعث ناخوانایی کدها میشه، فایل‌های با تعداد خطوط زیاد هستش. مثلا برنامه‌ای رو تصور کنید که فقط دارای یه فایل main هستش و شامل هفتصد، هشتصد خط کد هستش. کار کردن با چنین فایل‌های طولانی و پیمایش کردن بین اونها دشوار هستش. اگه بخوام اهمیت این موضوع رو روشن کنم بیاین یه مثال بزنیم. مطالعه‌ی یک کتابی که فقط شامل یک فصل هستش راحت‌تره یا کتابی که از بخش‌ها و فصل‌های مختلفی تشکیل شده؟ مسلما کتابی که محتوای اون به تناسب به بخش‌ها و فصل‌های مختلف دسته‌بندی شده راحت‌تر هستش. به طور مشابه کدهایی که به فایل‌های جداگانه تقسیم شده‌اند، کار کردن با اونها راحت‌تر هستش.

توصیه میشه که حتی الامکان کدهاتون رو به فایل‌های جداگانه‌ای بشکنید و در بخش‌های مورد نیاز اونها رو import کرده و استفاده کنید. این کار چند مزیت داره:

باعث خوانایی بهتر و پیمایش سریع‌تر در بین کدها میشه. این جوری دیگه خواننده مجبور نیستش برای پیدا کردن یه بخش خاص، کل فایل‌ها رو بگرده.

باعث رعایت اصل DRY یا Don’t Repeat yourself میشه. این یه اصل رایج هستش که به بیان ساده یعنی، حتی الامکان از کپی پیس کردن تکه کدها در سراسر برنامه‌تون پرهیز کنید. تعریف کردن توابع و کلاس‌های مختلف در فایل‌های جداگانه، باعث میشه که در سایر قسمت‌ها، دوباره مجبور به تعریف دوباره اونها نباشیم و کافیه که در فایل‌های دیگه با importکردن اونا و فراخوانی توابع مورد نظر برنامه‌مون رو تکمیل کنیم.

از نوشتن توابع طولانی پرهیز کنید.

یکی از موارد دیگه که باید از اون پرهیز کنیم، توابع طولانی هستند. حتی الامکان سعی کنید تعداد خط‌های توابع تون کم باشه و اگه برخی جاها احساس می‌کنید که توابع طولانی نوشتید، سعی کنید که خطوطی از توابع که در حال انجام یه کار خاصی هستند رو پیدا کنید و اون‌ها رو در توابع جدیدی تعریف کنید و در تابع اصلی تنها اونها فراخوانی کنید. بسته به نوع عملیاتی که انجام میدین ممکنه که پارامترهای ورودی و خروجی نیز برای اون زیر توابع تعریف کنید.

جمع بندی

در این قسمت از کتاب، درباره اهمیت زیبایی‌شناسی در کدها پرداختیم. همونطور که در ابتدای بخش نیز اشاره کردیم، در این لایه به محتوا و منطق کدها توجه نمی‌کنیم و تنها به مواردی ظاهری و بصری کدها اهمیت می‌دهیم. و اگه بخوام تمام نکات رو خلاصه کنم اینه که:

تا اونجایی که می‌تونید برنامه‌تون رو به بخش‌های مختلف بشکنید و از طولانی شدن تعداد خط‌های توابع و فایل‌ها جلوگیری کنید .به فاصله گذاری‌ها دقت داشته باشید و بخش‌های مختلف رو به واسطه‌ی فضای سفید از هم جدا کنید تا انسجام و تمایز بخش‌های مختلف حفظ بشه.

بخش پنجم

کامنت‌ها

در این بخش از کتاب قصد داریم تا به اهمیت کامنت‌ها و کار کردن با اونها بپردازیم. در این بخش به موضوعات زیر پرداخته میشه.

کامنت‌ها چی هستند و چه کاربردی دارند؟

بایدها و نبایدهای کامنت گذاری چیا هستند؟

چه جوری می‌تونیم با دونستن یه سری اصول درباره کامنت‌نویسی، کدهای خواناتری داشته باشیم؟

کامنت چیست؟

کامنت (Comment) یا توضیح، در زبان‌های برنامه‌نویسی، بخش‌هایی از برنامه هستند که توسط کامپایلر یا مفسر اجرا نمیشن و همونطور که از اسم‌شون هم مشخصه، نقش توضیحات رو برای ما دارند. در واقع زمان‌هایی پیش میاد که برنامه‌نویس قصد داره در بخش‌های مختلفی از کدها، یه سری توضیحات رو بنویسه. این توضیحات اگرچه توسط کامپایلر نادیده گرفته میشن و بود و نبودشون فرقی برای کامپیوتر نداره، اما وجود اونها به سایر برنامه‌نویسان کمک می‌کنه که درک بهتری از کدها داشته باشند.

یکی دیگه از کاربردهای کامنت‌ها، زمان‌هایی هستش که شما قصد دارین برای رفع اشکال یا دیباگ (Debug) برنامه‌تون، کاری کنید که خطوط مشخصی از کدهاتون اجرا نشه، یکی از راه‌هاش اینه که شما بیاین اون کدها رو پاک کنید، اما چون نوشتن مجدد اون بخش از کدها زمان‌بر هستش. شما در این مواقع می‌تونید به طور موقت اون کدها رو کامنت کنید تا بتونید برنامه‌تون رو رفع اشکال کنید و بعد از رفع شدن اون می‌تونید یا اون بخش از کدهای کامنت‌شده رو پاک کنید یا اونها را با اعمال یه سری تغییرات از حالت کامنت در بیارین.

ایده‌ی کلیدی کامنت نویسی

اگه بخوام یه ایده‌ی کلیدی برای فلسفه‌ی استفاده از کدها معرفی کنیم، می‌تونیم بگیم:

هدف از کامنت گذاری کمک کردن به خواننده درباره دانستن کد به اندازه نویسنده آن است.

توی کدنویسی، کسی که اون کد رو برای بار اول نوشته با کسی که قراره در آینده اون کدها رو مطالعه کنه، قطعا درک یکسانی از کارکرد برنامه ندارند. مسلما، اون فردی که کدنویس اولیه برنامه بوده قطعا یه سری پیش دانسته‌هایی بیشتری نسبت به سایر افراد داره. کامنت‌نویسی در چنین مواقعی کمک می‌کنه که اون برنامه‌نویس، سایر پیش‌فرض‌های ذهنی خودش رو هم به نوعی منتقل کنه. به این صورت که توسعه‌دهندگان بعدی اون برنامه درک نسبتا کاملی به اندازه توسعه دهنده اولیه داشته باشند.

چه زمان‌هایی نباید کامنت کرد

همونطور که اشاره کردیم، کامنت‌ها به ما کمک می‌کنند که درک کاملتری از برنامه داشته باشیم. اما لازمه که در استفاده از کامنت‌ها به چند تا نکته دقت داشته باشیم:

کامنت‌ها اگرچه بخش‌هایی از برنامه هستند که اجرا نمیشند اما میشه گفت که مثل سایر خطوط قابل اجرای برنامه هزینه‌بر هستند. منظور از هزینه چیه؟ یعنی مثل کدهای واقعی، فضا اشغال می‌کنند و باعث افزایش تعداد خطوط برنامه میشن. یا مثلا، یه هزینه‌ی دیگه‌ای هم که شامل میشه، زمان و انرژی‌ای هستش که برای خوندن و نوشتن اونها باید صرف بشه.

پس به طور کلی باید به این نکته دقت داشته باشیم که در استفاده از کامنت‌ها زیاده‌روی نکنیم و به اندازه از اونها استفاده کنیم. حالا توصیه چیه؟

ایده‌ی کلیدی: چه زمانهایی کامنت نکنیم؟

اگه بخوام بگم که چه زمان‌هایی نباید کامنت کنیم میشه گفت:

درباره حقایقی که به سرعت از خود کد می‌توانند استخراج شوند کامنت نکنید.

به بیان خودمونی، الکی کامنت نکنید. مثلا اگه می‌بینید یه سری از بخش‌های کدها، اونقدر ساده و واضح هستند که با خوندن همون خط از کدها میشه متوجه منطق و روال برنامه شد دیگه نباید برای اون بخش‌ها کامنت کرد چون عملا یه کاری بیهوده هستند. کاربرد صحیح کامنت‌ها، زمان‌هایی هستند که برنامه‌نویس‌های دیگه به تنهایی با خوندن کدهای برنامه نمی‌تونند به آسونی متوجه منطق و روال برنامه بشند. اینجا هستش که نیاز به کامنت داریم .

البته این نکته رو هم باید مد نظر داشت که به عنوان یه کدنویس، همواره باید سعی کنیم، تا اونجایی که میشه کدهامون رو ساده بنویسیم، به حدی که برای توضیحات بیشتر نیاز به کامنت نداشته باشیم.

مثلا توی انتخاب نام‌ها، به جای اینکه نام‌های بدی رو بنویسیم و برای توضیح بیشتر اون نام‌ها از کامنت‌ها استفاده کنیم، در عوض بیایم از همون اول یه نام قابل فهمی و صحیحی انتخاب کنیم که دیگه نیازی هم به کامنت نباشه.

ثبت افکار هنگام کدنویسی

یکی از کاربردهای کامنت‌نویسی، ثبت افکار هستش. مثلا در بخش‌هایی از برنامه، نیازه که یه سری نکات مهم، هشدارها و موارد خاصی ذکر بشن. چرا که دونستن چنین مواردی برای برنامه‌نویسان دیگه مهم هستش و ندونستن اونها ممکنه باعث به وجود آمدن مشکلاتی بشه.

کاربرد دیگر کامنت گذاری TODO

یکی از انواع کامنت‌های کاربردی، کامنت‌های TODO هستند. این کامنت‌ها معمولا در بخش‌هایی از کدها نوشته میشند که ما قراره بعدا به اونها رسیدگی کنیم. مثلا یه به تابعی رو تعریف می‌کنیم اما در حال حاضر قصد نداریم بدنه‌ی اونو تکمیل کنیم و در آینده این کار رو انجام میدیم. در این جور مواقع برای اینکه یادمون باشه که در آینده به این بخش از کدها مراجعه کنیم، از کامنت‌های TODO استفاده می‌کنیم. کامنت‌های TODO معمولا توسط

IDEها شناخته میشند و به ما کمک می‌کنند که در صورت نیاز، سریع‌تر به اون بخش‌ها دسترسی داشته باشیم.

پس به طور کلی هر جا از کدهاتون که نیاز شد بعدا سراغشون برید رو می‌تونید از کامنت‌های TODOاستفاده کنید.

خود را در کفش خواننده قرار دهید

آیا ممکن است سوال بپرسد؟ در مرحله اول کدها رو ساده تر کنید اگه نشد کامنت بذارین، البته کامنت‌های دقیق

پیش بینی سوالات احتمالی، پیش بنیین مشکلات احتمالی

به طور کلی اگه بخوایم یه توصیه‌ی عمومی در کامنت نویسی داشته باشیم، می‌تونیم بگیم که:

همواره خودتون رو جای خواننده‌های دیگه قرار بدین. در درجه‌ی اول تلاش کنید تا کدها رو به گونه‌ای بنویسید که برای توضیحات بیشتر نیازی به کامنت نداشته باشند و از خوندن مستقیم کدها بشه متوجه کدها شد. در مرحله‌ی بعد، اگه دیدین نیاز دارین تا برای یه سری بخش‌ها کامنت بنویسید سعی کنید که به این موارد دقت داشته باشید.

در نظر داشتن سوالات احتمالی

همیشه در هنگام کدنویسی سعی کنید این سوالات رو از خودتون بپرسید: آیا بعدا برای بقیه خوانندگان هنگام خوندن این بخش سوالی پیش نیم‌آید؟ آیا اینجا لازمه که اشاره کنم این بخش، یه بخش حیاتی و مهم هستش و مثلا برنامه‌نویس نباید اینجا رو تغییر بده؟و به طور کلی هرگونه سوالی که ممکنه بعدا پیش بیاد و یا هرگونه توضیح اضافه‌تری که احساس می‌کنید باید قرار داده بشه رو توی کامنت‌ها ذکر کنید.

تا حد ممکن کامنت‌هاتون رو خلاصه و دقیق بنویسید

همونطور که گفتیم، کامنت‌ها هزینه‌بر هستند، هم هزینه برای نوشتن اونها، هم هزینه برای خوندن و متوجه شدن اونها. پس تا اونجایی که می‌تونید سعی کنید در عین خلاصه و

کوتاه بودن کامنت‌ها، اونها رو به صـورت دقیق بنویسـید تا باعث به وجود اومدن ابهام هم نشه.

بخش ششم

ساده‌سازی جریان برنامه

در قسمت‌های گذشته پیرامون مسایلی برای ظاهر کدها و بخش بصری آنها صحبت کردیم. در ادامه‌ی سلسله مقالات قصد داریم تا به راهکارهایی سطح بالاتر از جمله ساده‌سازی منطق برنامه بپردازیم. برای مثال یکی از ویژگی‌های کدهای ناخوانا اینه که در اونها، از حلقه‌ها و شرط‌های تو در تویی استفاده شده که این امر باعث ناخوانا شدن کدها و گیج شدن خواننده میشه. در ادامه قصد داریم تا به شما راهکارهایی رو بدیم تا در زمینه‌ی ساده‌نویسی منطق و دستورات برنامه، موثرتر عمل کنید.

اگه بخوام در یک کلمه بگم که به صورت ایده‌آل چه کدی خوانا هستش می‌تونیم بگیم که کدهایی که فاقد هرگونه دستور شرطی یا حلقه و سایر دستورات کنترلی باشند، کدهای ساده‌ای هستند. چرا که این کدها صرفا خط به خط اجرا میشن و خواننده رو مجبور نمی‌کنند که برای درک اونها مجددا اون خطوط رو مطالعه کنه. اما خب نوشتن برنامه‌ای که فاقد هر گونه دستور کنترلی باشه سخته و در عمل غیر ممکنه. ولی تا اونجایی که ممکنه باید تلاش کنیم تا میزان جریان‌های کنترلی به حداقل برسه.

ایده‌ی کلیدی

تمام عبارات شرطی، حلقه‌ها، و دیگر تغییرات برای جریان کنترل را تا حد امکان طبیعی کنید، به حدی که طوری نوشته بشه که خواننده رو متوقف نکنه و مجبور به خواندن دوباره کد نشه.

به عبارت دیگه اگه هر چقدر کدهای شما روان‌تر بودند و کمتر باعث شدند که خواننده مکث کنه و اونها رو مجبور به خواندن دوباره کد کنه، این نشون دهنده‌ی این هستش که کدهای شما از خوانایی بالاتری برخوردارند .

در ادامه قصد داریم تا به بررسی چند راهکار در زمینه‌ی ساده‌سازی جریان کنترل کدها ارائه کنیم.

ترتیب آرگومان‌ها در عبارات شرطی

بیاین با یه مثال شـروع کنیم. از بین دو تکه کد ذکر شـده زیر به نظر شـما کدوم یک از اینها خواناتره.

```
if (length >= 10)
```

```
if (10 <= length)
```

به احتمال زیاد اولی خواناتر هستش. اما در مورد مثال بعدی چی فکر می‌کنید؟

```
while (bytes_received < bytes_expected)
```

```
while (bytes_expected > bytes_received)
```

در این مثال هم دوباره کد اول خواناتر هسـتش. پس اگه بخواهیم یه قانون کلی در ترتیب آرگومان‌ها ذکر کنیم، می‌تونیم بگیم:

معمولا متغیری که تصمیم گیری‌های ما بر اساس اون انجام میشه در ابتدا ذکر میشه.

ترتیب بلاک‌های if else

یکی دیگه از مواردی که می‌تونه در خوانایی کدها موثر باشه، ترتیب بلاک‌های شرطی if/else هسـتش. اگرچه در عمل و اجرا ترتیب دسـتورات شـرطی if/else تفاوت چندانی نداره، و شـاید برای شما هم ترتیب اونها اهمیت زیادی نداشته باشه اما اگه بخواهیم دقیق‌تر این مساله رو بررسـی کنیم، رعایت ترتیب صـحیح در نوشـتن بلاک‌های شـرطی می‌تونه در سـرعت خواندن تاثیرگذار باشه.

معمولا پیشـنهاد میشـه که بخش‌های سـاده‌تر و یا بخش‌هایی که مثبت هسـتند رو در ابتدا پردازش کنید. چرا که اینکار باعث افزایش سرعت خواندن میشه.

عملگر شرطی سه تایی

در برخی از زبان‌های مشابه c شما می‌تونید دستورات شرطی خودتون رو که دارای دو حالت true/false هستند به صورت زیر تعریف کنید.

```
condition ? a : b
```

در اینجا condition نشان‌دهنده‌ی یک عبارت منطقی هستش که می‌تونه درست یا نادرست باشه. در صورت صحیح بودن بخش a اجرا میشه و در صورت نادرست بودن بخش b. مثال:

```
time_str += (hour >= 12) ? "pm" : "am";
```

در اینجا بررسی می‌کنه که آیا ساعت ما بعد از ظهر هستش یا قبل از ظهر. طبعیتا نوشتن این مثال به کمک عملگر شرطی سه تایی نسبت به مثال زیر هم جمع و جور تر هستش و هم خواناتر.

```
if (hour >= 12) {
   time_str += "pm";
} else {
   time_str += "am";
}
```

اما آیا همیشه استفاده از عملگر شرطی سه تایی باعث خوانایی کد میشه. مثلا کد زیر رو در نظر بگیرید.

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa / (1 << -exponent);
```

Copy

به نظر شما، آیا این فشرده کردن کدها باعث کاهش خوانایی و سرعت نشده. برای مثال، تکه کد بالا رو میشه به صورت زیر بازنویسی کرد.

```
if (exponent >= 0) {
   return mantissa * (1 << exponent);
} else {
   return mantissa / (1 << -exponent);
}
```

در این مثال اگرچه تعداد خط‌ها بیشتر هستش اما سرعت خواندن بالاتره. پس اگه بخواهیم یه قاعده کلی برای استفاده یا عدم استفاده از عملگر شرطی سه تایی داشته باشیم، می‌تونیم بگیم:

در خواناتر کردن‌ها، به جای تلاش برای کاهش تعداد خط‌ها، برای کاهش زمان خواندن آنها کوشش کنید.

به بیان دیگه، اگه زمان مطالعه‌ی کد به حد بهینه‌ای کم شد، این نشون دهنده‌ی این هستش که کد شما خوانا هستش حتی اگر حجم اون کدها نسبتا زیاد بشه.

بلاک‌های تو در تو

یکی از مواردی که باعث کاهش سرعت خواندن کدها و افزایش سرباز ذهنی برای خواننده میشه، بلاک‌های تو در تو هستند. بلاک‌های تو در تو باعث ایجاد بار ذهنی بیشتری در ذهن خواننده میشه و ممکنه که حتی خواننده رو مجبور به دوباره خواندن کنه.

یکی از راهکارهایی که میشه برای رفع تو در تو بودن بلاک‌ها استفاده کرد، بازگشت سریع از توابع هستش. بعضی از برنامه‌نویسان تصور می‌کنند که توابع اونها نباید بیش از یک دستور returnداشته باشه و همین امر ممکنه باعث پیچیده و تو در تو شدن کدها بشه.

نکات پایانی

توی این پست درباره یه سری نکات درباره ساده‌سازی جریان برنامه گفتیم. به طور کلی در کدهای ما هر چقدر از دستورات کنترلی ساده‌تری استفاده بشه، سرعت خواندن رو افزایش میده. در خوانایی کدها، همواره تمرکزتون رو بر روی کاهش زمان خوندن کدها بذارین. ممکنه حجم کدهای شما زیاد بشه اما در این صورت از ابهام و پیچیدگی اونها کاسته میشه.

آکادمی متنو را در شبکه‌های اجتماعی دنبال کنید.

Site: learn.matno.co

Instagram: instagram.com/matnoacademy

LinkedIn: linkedin.com/company/matnoacademy

Aparat: aparat.com/matnoacademy

Virgool: virgool.io/@matnoacademy