۱۰ اصلی که هر برنامه نویسی باید از آن پیروی کند

همه برنامه نویس ها توانایی نوشتن کد را دارند اما چند نفر از آنها بر کارشان تسلط دارند؟

اگر با برنامه نویسی آشنایی داشته باشید احتمالا با مسائلی از قبیل کدهایی با کلاف های سردرگرم، زنجیره های if-else طولانی، ناکارآمد شدن برنامه با تغییر یک متغیر، توابع مبهم وغیره، بیگانه نیستید. بسیاری از برنامه نویسان کم تجربه هنگام توسعه نرم افزار با چنین مشکلاتی مواجه می شوند.

هنگام توسعه نرم افزار به طراحی برنامه ای که فقط کار کند بسنده نکنید، بلکه آن را به گونه ای طراحی کنید که توسط افراد دیگر هم قابلیت استفاده و پشتیبانی را داشته باشد. بدین منظور باید از اصول زیر پیروی کنید:

۱. ساده سازی

اصل ساده سازی در بسیاری از مسائل روزمره قابل استفاده است اما در طراحی پروژه های متوسط و بزرگ باید به صورت ویژه مورد نظر قرار گیرد.

این اصل را باید از مرحله اول توسعه برنامه که تعریف مقیاس پروژه است، مد نظر قرار دهید. برای مثال چون شما به طراحی بازی علاقه دارید قرارنیست یکی از بهترین و بزرگترین بازی های دنیا را طراحی کنید. برای اطمینان بیشتر، زمانی که احساس کردید پروژه را به اندازه کافی کوچک کرده اید، یک مرحله دیگرهم آن را تسهیل کنید.

در مرحله برنامه نویسی هم ازاین اصول ساده سازی پیروی کنید. طراحی و نوشتن الگوریتم ها و کدهای پیچیده به زمان بیشتری نیاز دارد، امکان وقوع باگ و خطا در آنها بیشتر است و اصلاح آنها هم دشوارتر خواهد بود. نویسنده مشهور فرانسوی آنتوان دوسنت اگزوپری این مساله را به صورتی زیبا شرح داده است: تکامل زمانی به دست نمی آید که چیز دیگری برای اضافه کردن باقی نمانده باشد، بلکه زمانی به دست می آید که هیچ چیز دیگری را نتوان حذف کرد.

۲. عدم تکرار

پیروی از این اصل باعث نوشتن کدهایی مرتب می شود که امکان اصلاح آنها در صورت وقوع باگ نیز ساده خواهد بود. بنابرین هنگام برنامه نویسی باید از تکرار داده ها و مسائل منطقی خودداری کنید. برای تعیین کدهای تکراری این سوال را از خود بپرسید: اگر قرار باشد این قسمت از برنامه را تغییر دهیم باید چه مقدار از کدها را اصلاح کنم؟

فرض کنید در حال توسعه اپلیکیشنی برای دسته بندی پادکست ها هستید. در صفحه جستجوی اپ، کد فراخوانی جزئیات پادکست را طراحی کرده اید. حال اگر تمام این موارد را در یک تابع خلاصه کنید هنگام اصلاح یا تغییر کد، کار شما تا ۶۰ درصد کمتر خواهد شد.

۳. برنامه باز و بسته

فارغ از اینکه در پروژه های خود از چه زبانی بهره می برید، کدها را باید به گونه ای بنویسید که امکان توسعه برنامه را برای کاربر فراهم کرده اما به وی اجازه اصلاح کدها را ندهد. این اصل هم باید در تمامی پروژه ها به ویژه هنگام انتشار کتابخانه ها یا فریمورک ها رعایت شوند.

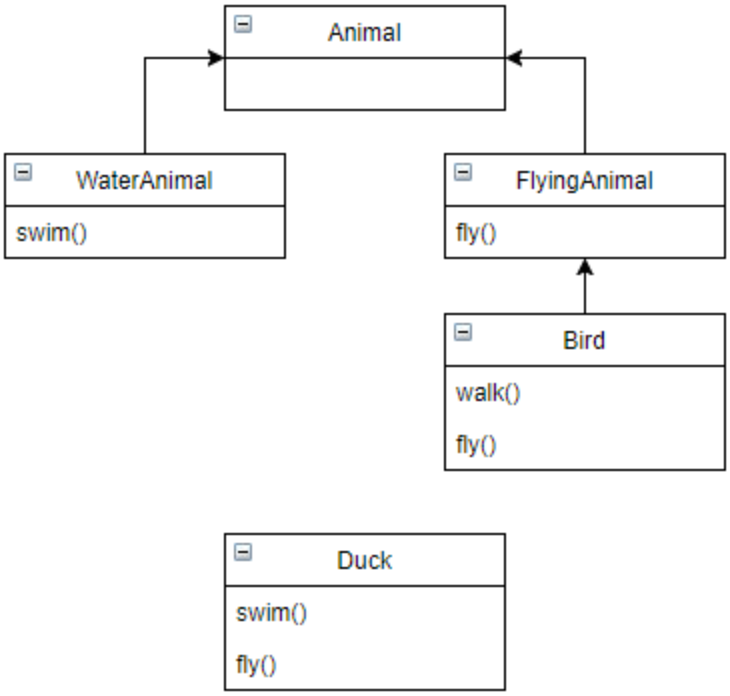
برای مثال فرض کنید که یک چارچوب GUI را به گونه ای منتشر می کنید که امکان اصلاح آن توسط کاربران فراهم باشد. در چنین حالتی پس از انتشار یک بروزرسانی مهم، کاربران برای بهره مند شدن از آن باید تمام تغییرات خود را مجددا پیاده سازی کنند که مسلما خوشایند آنها نخواهد بود.

برای جلوگیری از این مشکل باید با جلوگیری از امکان اصلاح کدها، کاربران را به توسعه برنامه تشویق کنید. با این کار هسته اصلی را از تغییر مصون نگاه داشته و به ثبات و پایداری برنامه کمک می کنید. این اصل در توسعه یک API خوب اهمیت زیادی دارد.

۴. ترکیب در وراثت

اصل ترکیب در وراثت بدین معنی است که اشیاء با رفتار پیچیده باید شامل اشیاء با رفتار یکتا بوده و در آنها از اضافه کردن رفتارهای جدید یا به ارث بردن یک کلاس خودداری شود.

عدم استفاده از این اصل می تواند به دو مشکل بزرگ منجر شود، اول سلسله مراتب وراثت ممکن است در یک چشم بر هم زدن به کلافی سردرگرم تبدیل شود. دوم اینکه هنگام تعریف رفتارهای خاص و پیاده سازی رفتار از یک شاخه وراثت در شاخه ای دیگر انعطاف پذیری کمتری خواهید داشت.



از مزایای ترکیب می توان به نظم بیشتر، پشتیبانی ساده و انعطاف پذیری بالا در تعریف انواع مختلف رفتارها اشاره کرد. هر رفتار یکتا کلاس خاصی است و با ترکیب رفتارهای یکتا می توانید رفتارهای پیچیده تری بسازید.

۵. مسئولیت پذیری یکتا

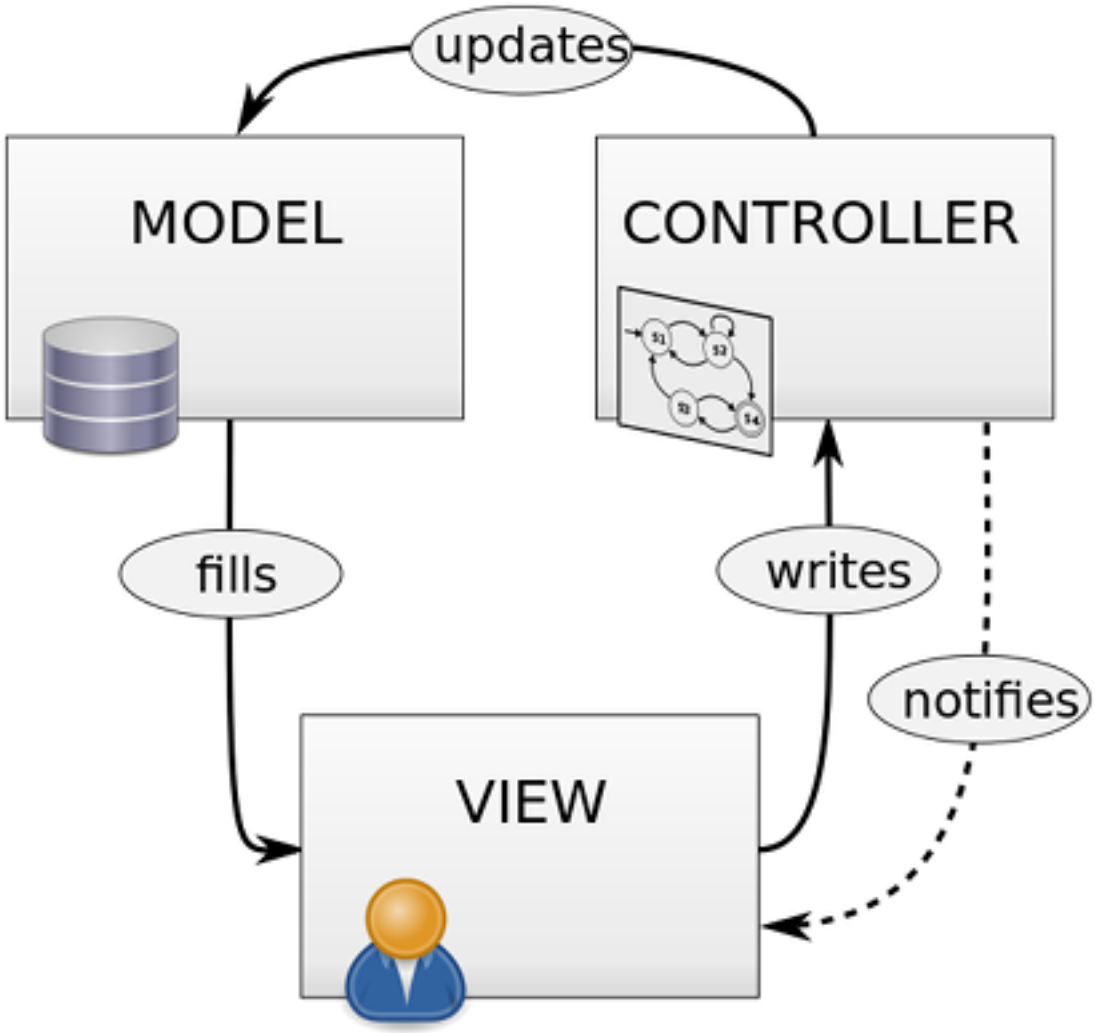
اصل مسئولیت پذیری یکتا بدین معنی است که هر کلاس یا ماژول برنامه باید وظیفه ای خاص را بر عهده داشته باشد. روبرت مارتین هم در این باره می گوید: هرکلاس باید تنها به یک دلیل تغییر داده شود.

اغلب کلاس ها و ماژول ها در ابتدا از این ویژگی برخودار هستند اما با اضافه شدن ویژگی ها و رفتارهای جدید به ابرکلاس هایی تبدیل می شوند که ممکن است شامل صدها یا حتی هزاران خط کد باشند. اگر این کلاس ها بیش از حد بزرگ شوند باید آنها را به واحدهایی کوچکتر تقسیم کنید.

۶. جداسازی دغدغه ها

این اصل هم مثل مورد قبلی است اما در سطحی انتزاعی تر صورت می گیرد. بر این اساس یک برنامه باید از تعداد زیادی Encapsulationهای غیرهمپوشان طراحی شود که از محتوای یکدیگر بی خبر باشند. معماری نرم‌افزار MVC، مثال خوبی از این اصل است که در آن برنامه به سه ناحیه مجزای داده (مدل)، منطق (کنترلر)

و آنچه که کاربر نهایی می بینید (نما) تقسیم می شود. این معماری در بسیاری از فریم ورک های وب محبوب به کار برده شده است.



برای مثال کدی که مسئول بارگذاری و ذخیره داده ها در دیتابیس است به اطلاع از چگونگی رندر داده ها در وب نیازی ندارد. علاوه بر این کد رندر هم پس از دریافت ورودی از کاربر نهایی داده ها را به برای پردازش به کد کنترلر ارسال می کند. بدین ترتیب هرکد تنها وظیفه خود را انجام داده و در مسائل غیرمرتبط درگیر نمی شود.

نتیجه استفاده از این معماری برخورداری از کدی ماژولار است که پشتیبانی را تسهیل می کند. علاوه بر این در آینده هم اگر نیاز به تغییر یک بخش خاص داشته باشید بدون نگرانی از مسائل دیگر می توانید روی اصلاح بخش مورد نظر تمرکز کنید.

۷. عدم نگرانی از آینده

بنابر این اصل، شما هنگام نوشتن یک کد هرگز نباید به این فکر کنید که ممکن است در آینده هم به آن نیاز پیدا کنید چرا که با این کار تمرکز خود را از دست داده و ممکن است با اضافه کردن موارد غیر ضروری کدها را پیچیده تر کنید.

اغلب برنامه نویس های کم تجربه به دنبال نوشتن ساده ترین و منطقی ترین کدها هستند اما افراط در این کار به توسعه کدهایی منجر می شود که تغییر، اصلاح و توسعه آنها در آینده دشوار خواهد بود.

راه حل غلبه براین مشکل این است که کدها را بررسی کنید و هرجا به الگوهای تکراری برخورد کردید آنها را ساده سازی کنید، اما هرگز پیش از نوشتن کامل کدها در مورد تکرار شدن یک کد خاص پیش داوری نکنید.

۸. از بهینه سازی پیش از موعد خودداری کنید

این اصل هم به مورد قبلی شباهت زیادی دارد اما تفاوت بین آنها این است که مورد هفتم به پیاده سازی الگوریتم های تکراری می پردازد اما این اصل به افزایش سرعت الگوریتم هایی می پردازد که ممکن است اصلا کند نباشند.

مشکل بهینه سازی پیش از موعد این است که شما تا پیش از توسعه هر قسمت، از کند بودن بخشی از آن مطلع نمی شوید. گرچه ممکن است برخی مواقع در این باره حدس بزنید و حدس شما هم درست از آب دربیاید اما با توجه به گستردگی کدها ممکن است در اغلب موارد اشتباه کنید. در این صورت با تلاش برای افزایش سرعت عملکرد تابعی که اصلا کند نیست یا نقش چندانی در برنامه ندارد، زمان ارزشمند خود را هدر می دهد.

بنابرین ابتدا بر طبق برنامه ریزی پیش رفته و پس از توسعه هر بخش به مشکلات احتمالی رسیدگی کنید.

۹. بهینه سازی، بهینه سازی، بهینه سازی

یکی از سخت ترین حقایقی که برنامه نویسان کم تجربه باید قبول کنند این است که به ندرت یک برنامه در همان سری اول به صورت صحیح عمل می کند.

اضافه کردن قابلیت های ویژه ممکن است بسیار خوشایند به نظر برسد اما این قابلیت ها باعث افزایش پیچدیگی کلی شده و ممکن است بر کارایی بخش های قبلی تاثیر منفی بگذارند.

بنابرین کدهای یک برنامه در ابتدا به بازبینی، بازنویسی و حتی طراحی مجدد نیاز دارند. شما در ابتدای طراحی برنامه ممکن است از قابلیت ها و بخش های مختلف آن اطلاع نداشته باشید اما پس از پایان کار از این موارد اطلاع دقیق دارید و با توجه به این مساله باید کدهای ابتدایی را هم بازبینی کنید.

توجه داشته باشید که در این فرایند لازم نیست قابلیت های جدیدی را به کدها اضافه کنید و تنها کافی است که موارد غیر ضروری را از آنها حذف کنید.

۱۰. کد مرتب = کد هوشمند

در برنامه نویسی کد هوشمند به معنای کد پچیده نیست بلکه کدی است که علاوه بر کارایی بالا برای دیگر برنامه نویس ها هم قابل درک باشد. بنابرین هنگام نوشتن یک برنامه سعی نکنید با پیچیده کردن آن نبوغ و استعداد خود را به رخ بکشید چون در واقع کسی به این موضوع اهمیت نمی دهد .

نمونه ای از یک کد هوشمند تجمیع منطق های مختلف در یک خط کد یا استفاده از پیچدیگی های زبان برای نوشتن کدهای ساده ولی کارآمد است.

برنامه نویس های موفق و کدهای خوانا به سرعت مطرح می شوند. بنابرین برای افزایش خوانایی کدها در مواقع لازم برای توضیح نکات از کامنت استفاده کنید، به راهنمای برنامه توجه کنید و از نوشتن کدهای جاوا در پایتون یا بالعکس خودداری کنید.

یک برنامه نویس خوب چه ویژگی هایی دارد؟

اگر این سوال را از پنج نفر بپرسید، ده پاسخ مختلف دریافت می کنید. از نظر من یک برنامه نویس خوب کسی است که می داند کدنویسی باید در خدت کاربر نهایی باشد، کار کردن با او در یک تیم ساده است و همیشه پروژه ها را با ویژگی های مشخص شده و به موقع تحویل می دهد.