۱۰ قانون که هر برنامه نویسی باید دنبال کند

همه برنامه نویسان قادر به کدنویسی هستند اما آیا توانایی همه در یک حد است؟

همه ما داستان های ترسناکی را درباره کد اسپاگتی، زنجیره های بزرگ if-else، برنامه هایی که به خاطر یک متغیر کوچک از کار می افتند و توابعی که مبهم به نظر می رسند شنیده ایم. اما این اتفاقات زمانی رخ می دهد که شما با تجربه ناکافی به دنبال تولید برنامه ای بزرگ هستید.

شما نباید به نوشتن کدی که کار کند اکتفا کنید بلکه هدفتان باید تولید محصولی باشد که برای دیگران هم قابل درک بوده و حفظ و نگهداری آن ساده باشد. بدین منظور باید از ۱۰ اصل زیر پیروی کنید تا محصول نهایی تمیز و سر راست باشد.

۱ .قانون KISS

کلمه KISS مخفف Keep it Simple, Stupid است. اصل هیچ چیز رو پیچیده نکن در بسیاری از جنبه های زندگی کاربرد دارد اما پیروی از آن در پروژه های با مقیاس متوسط تا بزرگ ضروری است.

این فرایند از ابتدا و حین تعریف مقیاس محصول شروع می شود. صرف داشتن علاقه به توسعه بازی بدین معنی نیست که شما بازی GTA بعدی را به بازار عرضه می کنید. زمانی که فکر می کنید همه چیز به اندازه کافی ساده شده، یک مرحله دیگر پروژه را ساده کنید.

پس از شروع کدنویسی هم باید این اصل را در دستور کار قرار دهید. طراحی و نگارش کدهای پیچیده به زمان بیشتری نیاز دارد، امکان وقوع باگ و خطا در آن بیشتر بوده و تصحیح آنها سخت تر است. به قول آنتوان دو سنت اگزوپری «کمال زمانی رخ می دهد که چیز دیگری برای حذف وجود ندارد نه زمانی که چیزی برای اضافه کردن باقی نمانده است.»

۲ .قانون DRY

کلمه DRY مخفف Don't Repeat Yourself است. لازمه نگارش کدهای تمیز و با قابلیت اصلاح آسان، پیروی از اصل اجتناب از دوباره کاری است. حین کدنویسی باید از کپی داده ها و منطق برنامه خودداری کنید چرا که در این صورت این اصل را نقض کرده اید. تکرار کدها به معنی هدر دادن زمان است و یکی از بهترین راه ها برای تشخیص این اشتباه پرسیدن این سوال از خودتان است: برای تغییر کاربرد برنامه چه میزان از کدها را باید اصلاح کنم؟

فرض کنید در حال توسعه اپلیکیشن دایرکتوری پادکست هستید. در صفحه جستجو کدی برای فراخوانی جزییات پادکست دارید. در صفحه پادکست هم کدی برای فراخوانی جزییات پادکست دارید. در صفحه موارد مورد علاقه نیز همین کد تکرار شده است. با یکپارچه کردن این موارد در یک تابع، اصلاح همگی آنها در آینده تنها به یک بار ویرایش نیاز خواهد داشت.

۳.قانون باز / بسته

فارغ از اینکه در حال برنامه نویسی به زبان پایتون یا جاوا هستید باید به گونه ای این کار را انجام دهید که کد در برابر اصلاح، نفوذ ناپذیر و در عین حال آماده توسعه باشد. این اصل در همه موارد کاربرد دارد اما حین انتشار کتابخانه یا فرمورک برای دیگران اهمیت فوق العاده ای پیدا می کند.

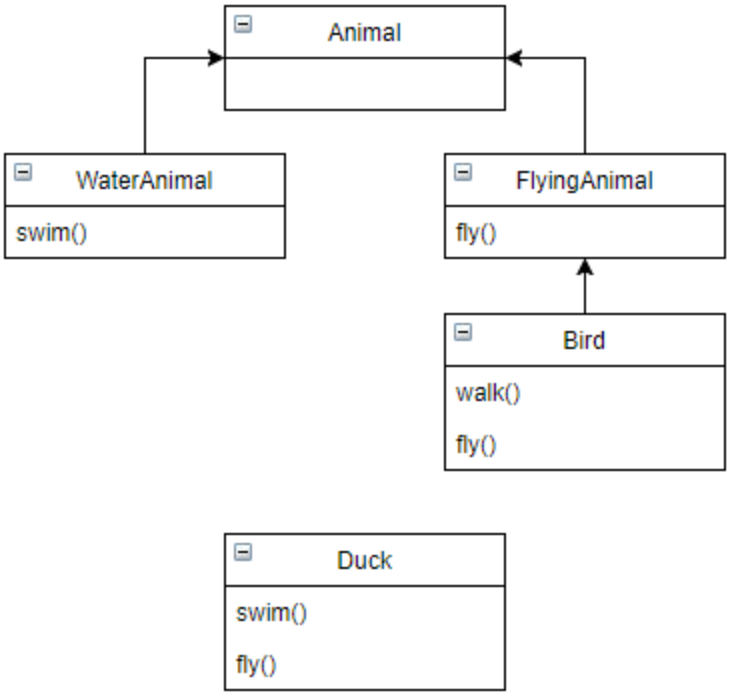
برای مثال فرض کنید یک پروژه فریمورک GUI را در دست اجرا دارید، شما می توانید همانطور که هست آن را منتشر کرده و انتظار داشته باشید کاربران نهایی آن را اصلاح کرده و در پروژه هایشان به کار بگیرند. اما چهار ماه بعد که یک به روزرسانی قابل توجه را برای آن منتشر می کنید چه اتفاقی رخ می دهد؟ چگونه کاربر باید بدون دور انداختن تمام کارهایی که انجام داده موارد جدید را به پروژه اضافه کند؟

از سوی دیگر اگر کدها را به گونه ای منتشر کنید که از انجام اصلاحات جلوگیری کرده و کاربر را به سوی افزونه ها سوق دهد،stability پایداری (جلوگیری از ایجاد مشکل در کارایی کد) وmaintainability قابلیت نگهداری (محدود شدن دغدغه کاربر به افزونه) از کدها بیشتر می شود. بنابرین اصل باز یا بسته در تولید یک API مطلوب ضروری است.

۴. قانون اولویت ترکیب بر وراثت

اصل ترکیب را بر وراثت ترجیح بده بدین معنی است که اشیا با رفتار پیچیده باید از اشیا با رفتارهای جداگانه تشکیل شوند و از ارث بردن کلاس و اضافه کردن رفتارهای جدید در آنها خودداری شود.

وابستگی بیش از حد در وراثت ممکن است به دو مشکل مهم منجر شود. نخست سلسله مراتب وراثت در یک چشم بر هم زدن بیش از حد پیچیده می شود و در ثانی انعطاف پذیری برای تعریف رفتارهای خاص کاهش می یابد به ویژه زمانی که در پی پیاده سازی رفتاری از یک شاخه وراثت در شاخه ای دیگر هستید.



نوشتن ترکیب به مراتب سر راست تر بوده و نگهداری از آن ساده تر است. علاوه بر این انعطاف پذیری بسیار بالایی را بسته به نوع رفتاری که برنامه نویس تعریف می کند، فراهم می سازد. هر رفتار مجزایی کلاس خاص خوش را دارد و با ترکیب این رفتارهای مجزا می توانید موارد پیچیده ای خلق کنید.

۵ .قانون مسئولیت پذیری واحد

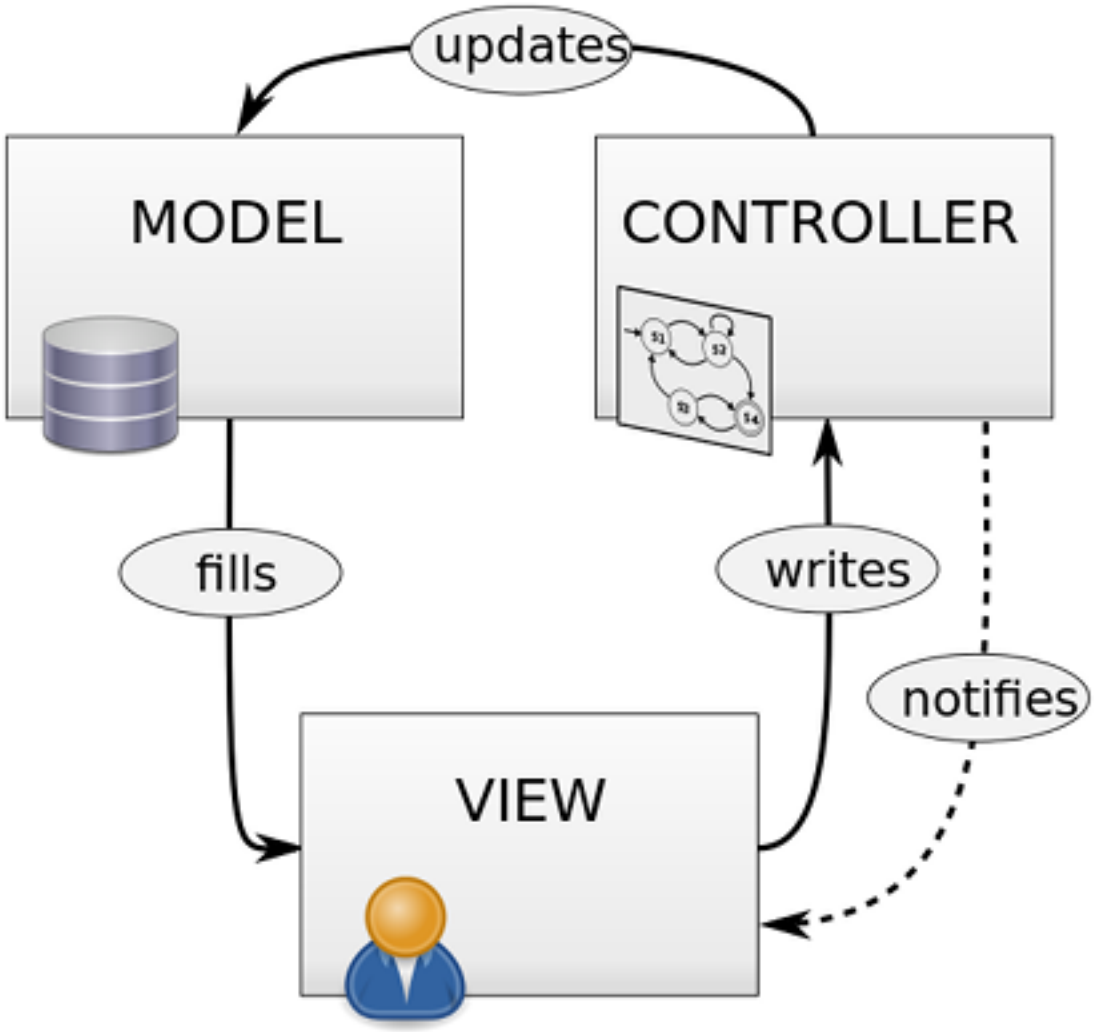
این اصل تاکید می کند که هر کلاس یا ماژول در یک برنامه باید تنها دغدغه مسئولیت خودش را داشته باشد. رابرت سی مارتین در این مورد می گوید : «یک کلاس باید تنها یک دلیل برای تغییر داشته باشد.»

کلاس ها و ماژول ها اغلب به همین ترتیب شروع می شوند اما با اضافه شدن ویژگی ها و رفتارهای جدید، پیش از آنچه فکرش را بکنید به مواردی با صدها و حتی هزاران خط کد تبدیل می شوند. در این صورت باید آنها را به کلاس ها و ماژول های کوچکتر خرد کنید.

۶ .قانون جداسازی دغدغه ها

این اصل هم شباهت زیادی به مسئولیت پذیری واحد دارد با این تفاوت که در یک سطح انتزاعی بالاتر قرار می گیرد. یک برنامه باید بر مبنای non-overlapping encapsulations طراحی شود و این encapsulation ها نباید از یکدیگر آگاه باشند.

یک مثال خوب از این اصل الگوی MVC است که برنامه را به سه ناحیه تقسیم می کند: داده یا مدل، منطق یا کنترلر و آنچه که کاربر نهایی می بیند یا نما. امروزه در بیشتر فریمورک های وب از انواع مختلف MVC استفاده می شود.



برای مثال کدی که بارگذاری و ذخیره داده در دیتابیس را مدیریت می کند نیازی به اطلاع از چگونگی رندر داده در وب ندارد. شاید کد رندرینگ، ورودی را از کاربر بگیرد اما برای پردازش آن را به کد منطق تحویل میدهد.

نتیجه پیروی از این اصل توسعه کدی ماژولار است که فرایند نگهداری را ساده تر می سازد. در آینده نیز در صورت لزوم بازنویسی تمام کدهای رندرینگ، در مورد نحوه ذخیره و پردازش داده دغدغه ای نخواهید داشت.

۷ .قانون کدنویسی در زمان حال

این قانون بر این ایده استوار است که شما نباید برای عملکردی که ممکن است در آینده به آن نیاز داشته باشید کدنویسی کنید چرا که در صورت عدم نیاز به آن، تنها وقتتان را هدر داده اید و علاوه بر این پیچیدگی و حجم کدها را بی دلیل بیشتر کرده اید.

این مورد را می توان حالت خاصی از اصل اول و در عین حال پاسخی به آنها که اصل اجتناب از تکرار را بیش از حد جدی گرفته اند به شمار آورد. اغلب برنامه نویسان کم تجربه به سختی به دنبال نوشتن کد در کلی ترین و چکیده ترین حالت ممکن هستند اما این کار ممکن است نگهداری از کد را غیرممکن سازد.

ترفندی که باید به کار ببرید این است که تنها در موارد لازم از تکرار خودداری کنید. هر زمان متوجه شدید که چندین بار بخش زیادی از کد را کپی کرده اید آن وقت دست به کار شوید و هرگز احتمال تکرار کدها را پیش بینی نکنید چرا که اغلب این اتفاق رخ نمی دهد.

۸ .قانون اجتناب از بهینه سازی زودهنگام

این اصل هم بی شباهت به مورد قبل نیست و تفاوت آنها در این است که مورد هفتم به تمایل برای پیاده سازی رفتار غیرضروری می پردازد اما اینجا در مورد تمایل به افزایش زودهنگام سرعت الگوریتم ها صحبت می کنیم.

مشکل بهینه سازی زودهنگام در این است که تا زمان وقوع یک گلوگاه نمی توان از مکان رخداد آن باخبر شد. البته می توان این موارد را حدس زد و شاید گاهی اوقات هم درست از آب درآید اما اغلب اوقات با تلاش برای افزایش سرعت تابعی که چندان هم کند نیست تنها زمان ارزشمند خود را هدر می دهید.

ابتدا سعی کنید به ضرب الاجل های تعیین شده برسید سپس سراغ شناسایی گلوگاه ها بروید.

۹ .بازنویسی، بازنویسی، بازنویسی

یکی از سخت ترین نکاتی که برنامه نویسان تازه کار باید با آن کنار بیایند این است که به ندرت کدها همان مرتبه اول به درستی عمل می کنند. شاید در مراحل اولیه کدنویسی همه چیز درست به نظر برسد اما با افزایش پیچیدگی برنامه احتمالا ویژگی های جدیدتر روی کارایی موارد قبلی تاثیر گذاشته و با آنها تداخل پیدا می کنند.

پایه های کد به صورت پیوسته در حال تکامل هستند و بازبینی، بازنویسی و حتی باز طراحی بخش بزرگی از آنها، نه تنها طبیعی است بلکه توصیه می شود حتما این کار را انجام دهید. با گذشت زمان اطلاعات شما در مورد نیازهای پروژه بیشتر شده و با استفاده از این داده ها می توانید کدها را پیرایش کنید.

به خاطر داشته باشید که نیازی نیست زمان زیادی را صرف این کار کنید. یکی از شعارهای سازمان «Boy Scouts of America» با این اصل تناسب زیادی دارد: اردوگاه را تمیزتر از زمان وردتان ترک کنید. هر زمان که احساس کردید کد نیاز به بررسی یا اصلاح دارد، آن را سرراست تر کرده و بهینه سازی کنید.

۱۰ .قانون ارجح بودن کد تمیز بر کد هوشمند

حالا که صحبت از کد هوشمند شد بهتر است غرورتان را کنار گذاشته و طرز تفکرتان را عوض کنید. اگر شما هم فکر می کنید کد هوشمند کدی است که به جای راه حل، شبیه معما به نظر رسیده و مهارت برنامه نویس را به رخ بکشد، بهتر است بدانید کسی به آن اهمیت نخواهد داد.

یک نمونه از کد هوشمند قرار دادن هرچه بیشتر منطق در یک خط از برنامه است. نمونه دیگر آن استفاده از پیچیدگی های زبانی برای نوشتن کدهای عجیب ولی کاربردی است. در واقع از نظر برخی برنامه نویسان هر کدی که بتواند دیگران را به تعجب وادارد یک کد هوشمند است.

با این حال این کدهای واضح و توسعه دهندگان آنها هستند که به شهرت می رسند. از این رو بهتر است حین کدنویسی هرجا که لازم شد توضیحات ضروری را ارائه کرده و از دستورالعمل ها پیروی کنید. همچنین برای هرچه

تمیزتر شدن برنامه از نوشتن کدهای جاوا در قالب پایتون یا برعکس خودداری کرده و اصول ارائه شده در این مطلب را مد نظر قرار دهید.

برنامه نویس خوب چه ویژگی هایی دارد؟

اگر این سوال را از ۵ نفر بپرسید ۱۰ جواب مختلف دریافت می کنید. از نظر من برنامه نویس خوب کسی است که می داند کدنویسی در نهایت باید در خدمت کاربر نهایی باشد، همکاری با او در تیم آسان است و پروژه را بر اساس الزامات و سر وقت تمام می کند.

اگر برنامه نویسی را تازه شروع کرده اید چندان نگران این موارد نباشید و در عوض روی یادگیری کدنویسی بدون استرس تمرکز کنید.