

**97 چیز که هر برنامه­نویسی باید بداند**

ویرایش شده توسط کِولین هِنی

هدیه ای متن باز از ساسان صفری به کامیونیتی برنامه نویسی فارسی

**پیشگفتار**

*جدیدترین کامپیوتر می تواند با سرعت، قدیمی ترین مشکل در روابط بین بشریت را ترکیب کند و در پایان، رابط با مشکل قدیمی مواجه خواهد شد که چه بگوید و چگونه آن را بگوید.*

-ادوارد آر. مارو

در ذهن برنامه­نویسان چیزهای زیادی می­گذرد: زبان‌های برنامه‌نویسی، تکنیک‌های برنامه‌نویسی، محیط‌های توسعه، سبک کدنویسی، ابزارها، فرآیند توسعه، مهلت تحویل پروژه­ها، جلسات، معماری نرم‌افزار، الگوهای طراحی، پویایی تیم، کد، پیش­نیازها، اشکالات، کیفیت کد و چیزهای بسیار زیاد دیگر.

هنر، مهارت و علمی درمورد برنامه نویسی وجود دارد که بسیار فراتر از کدنویسی است. عمل برنامه نویسی دنیای گسسته رایانه ها را با دنیای سیال امور انسانی پیوند می دهد.

با توجه به چیزهای زیادی که برای دانستن وجود دارد، کارهای زیادی که باید انجام داد، و راه‌های بسیار زیاد برای انجام این کارها، هیچ شخص یا منبع واحدی نمی‌تواند ادعای «تنها یک راه درست» داشته باشد. در عوض، *97 چیز که هر برنامه نویس باید بداند*، از خرد جمعی و افراد باتجربه استفاده می کند تا تصویری هماهنگ و کلی از برنامه­نویسی را به برنامه­نویسان نشان دهد. مطالب این کتاب از توصیه های مبتنی بر کد تا فرهنگ، استفاده از الگوریتم تا تفکر اجایل، دانش پیاده سازی تا حرفه ای بودن، از سبک تا ماهیت، متغیر هستند.

دستاوردها مانند قطعات چندبخشی به هم متصل نیستند، و هیچ قانوني وجود ندارد که باید به هم متصل باشند - اگر چیزی باشد، عکس آن صادق است. ارزش هر دستاورد از متمایز بودن آن ناشی می شود. ارزش مجموعه در این است که چگونه دستاوردها مکمل، تاییدکننده و حتی در تضاد با یکدیگر هستند. هیچ روایت جامعی وجود ندارد: این وظیفه شماست که به آنچه می‌خوانید پاسخ دهید، درباره آن تأمل کنید، و آن‌ها را با هم مرتبط و با زمینه، دانش و تجربه‌ی خود مقایسه کنید.

**با احتیاط عمل کنید**

*سب رز*

*هر کاری انجام می دهید، با احتیاط عمل کنید و عواقب آن را در نظر بگیرید.*

*آنون*

مهم نیست در ابتدای کار یک برنامه زمانی چقدر راحت به نظر می رسد، برخی مواقع نمی توانید تحت فشار قرار نگیرید. اگر متوجه شدید که باید بین «انجام درست کار» و «انجام سریع آن» یکی را انتخاب کنید، اغلب جذاب است که «آن را سریع انجام دهید» با این درک که بعداً برمی‌گردید و آن را درست می‌کنید. وقتی این قول را به خود، تیم و مشتری خود می دهید، واقعا از ته دل­تان این قول را داده­اید. اما اغلب اوقات، نسخه بعدی، مشکلات جدیدی را به همراه دارد و شما روی آنها متمرکز می شوید. این نوع کار معوق به عنوان بدهی فنی شناخته می شود. مارتین فاولر در کتاب واژه­شناسی خود این نوع بدهی فنی را بدهی فنی عمدی می­نامد اما نباید آن را با بدهی فنی غیرعمدی اشتباه گرفت.

بدهی فنی مانند وام است: در کوتاه مدت از آن سود می برید، اما باید بهره آن را به طور کامل پرداخت کنید. میانبرهای موجود در کد، افزودن ویژگی ها یا اصلاح مجدد کد را دشوارتر می کنند و محل پرورش عیوب و تست­های دردسرساز هستند. هر چه دیرتر به آن رسیدگی کنید، اوضاع بدتر می شود. زمانی که به اصلاح اصلی برسید، ممکن است علاوه بر مشکل اصلی، مجموعه کاملی از انتخاب‌های طراحی نه چندان درست وجود داشته باشد که اصلاح و تصحیح کد را بسیار سخت‌تر می‌کند. در واقع، اغلب تنها زمانی که اوضاع آنقدر بد شده است که باید مشکل اصلی را برطرف کنید، واقعاً برای رفع مشکلات به عقب برمی‌گردید؛ ولی دیگر برطرف کردن آن آنقدر سخت می­شود که واقعاً نمی توانید زمان از دست بدهید یا خطری را متحمل شوید.

مواقعی وجود دارد که برای رسیدن به زمان تحویل یا اجرای بخش کوچکی از یک ویژگی، باید بدهی فنی داشته باشید. سعی کنید در این موقعیت قرار نگیرید، اما اگر شرایط کاملاً ایجاب می کند، انجام این کار مشکلی ندارد. اما (و این یک اما بزرگ است) باید بدهی فنی را ردیابی کنید و به سرعت آن را جبران کنید، در غیر این صورت همه چیز به سرعت نابود می شود. به محض اینکه تصمیم به انجام این کار گرفتید، یک کارت تسک بنویسید یا آن را در سیستم ردیابی مشکل خود وارد کنید تا مطمئن شوید که آن را فراموش نمی­کنید.

اگر جبران بدهی را در نسخه بعدی برنامه ریزی کنید، هزینه کم­ترین مقدار ممکن خواهد بود. پرداخت نشدن بدهی باعث تعلق سود خواهد شد و این سود باید برای قابل مشاهده بودن هزینه پیگیری شود. این امر بر تأثیر ارزش تجاری بدهی فنی پروژه تأکید و امکان اولویت بندی مناسب جبران را فراهم می کند. انتخاب نحوه محاسبه و پیگیری سود به پروژه بستگی دارد، اما باید آن را پیگیری کنید.

بدهی فنی را در اسرع وقت پرداخت کنید. در غیر این صورت بی احتیاطی کرده­اید.

**از اصول برنامه نویسی تابعی استفاده کنید**

*ادوارد گارسِن*

برنامه نویسی تابعی اخیراً مورد توجه جامعه برنامه نویسی قرار گرفته است. بخشی از آن به این دلیل است که ویژگی‌های نوظهور پارادایم تابعی برای حل کردن چالش‌های ناشی از تغییر صنعت به سمت چند هسته‌ای شدن هستند. با این حال علت اینکه این بخش شما را به دانستن برنامه‌نویسی تابعی توصیه می‌کند، این نیست.

تسلط بر پارادایم برنامه نویسی تابعی تا حد زیادی می تواند کیفیت کدهایی را که در زمینه های دیگر می نویسید، بهبود بخشد. اگر پارادایم تابعی را عمیقاً درک کرده و به کار ببرید، طرح­های شما درجه بسیار بالاتری از شفافیت ارجاعی را نشان خواهند داد.

شفافیت ارجاعی ویژگی بسیار مطلوبی است: یعنی توابع به طور مداوم نتایج یکسانی را با توجه به ورودی یکسان، صرف نظر از مکان و زمان فراخوانی، به دست می‌آورند. یعنی ارزیابی عملکرد کمتر -در حالت ایده آل، اصلاً- به عوارض جانبی mutable state بستگی دارد.

یکی از دلایل اصلی نقص در کد دستوری به متغیرهای قابل تغییر نسبت داده می شود. همه کسانی که این مطلب را می خوانند، بررسی کرده اند که چرا یک مقدار در یک موقعیت خاص آنطور که انتظار می رود نیست. آگاهی درمورد قابل مشاهده بودن یا نبودن متغیرها می تواند به کاهش این عیوب خطرناک یا حداقل شدیدا محدود کردن مکان آنها کمک کند، اما عامل واقعی آن­ها ممکن است در واقع طرح هایی باشد که از تغییرپذیری بیش از حد استفاده می کنند.

و مطمئناً در این زمینه کمک زیادی از صنعت دریافت نمی کنیم. مقدمه‌هایی برای شی‌گرایی تا حدودی چنین طراحی را ترویج می‌کنند، زیرا اغلب نمونه‌هایی متشکل از نمودارهایی از اشیاء با عمر نسبتاً طولانی را نشان می‌دهند که با رضایت، متدهای mutator را درون یکدیگر فراخوانی می­ کنند؛ این نوع فراوانی می‌تواند خطرناک باشد. با این حال، با طراحی هوشمندانه تست محور، به ویژه زمانی که مطمئن باشید نقش‌ها را شبیه­سازی می­کنید، نه اشیاء را، تغییرپذیری غیرضروری را می‌توان از بین برد.

نتیجه نهایی، طرحی است که معمولاً دارای تخصیص مسئولیت بهتر با توابع متعدد و کوچکتر است که به جای ارجاع به متغیرهای عضو قابل تغییر، بر روی آرگومان­های ارسال شده به آن­ها عمل می کند. نقص‌های کمتری وجود خواهد داشت، به‌علاوه، دیباگ کردن آن‌ها اغلب ساده‌تر خواهد بود، زیرا پیدا کردن مکان یک مقدار rogue تعریف شده، آسان‌تر از استنباط کانتکست خاصی است که منجر به نسبت­دهی اشتباه می‌شود. این کار به میزان شفافیت ارجاعی می‌افزاید، و مطمئنا هیچ چیز به اندازه یادگیری یک زبان برنامه‌نویسی کاربردی، که این مدل محاسبه در آن معمول است، این ایده‌ها را عمیقا وارد ذهن شما نمی کند.

البته این رویکرد در همه شرایط بهینه نیست. برای مثال، در سیستم‌های شی‌گرا، این سبک اغلب نتایج بهتری را با توسعه مدل دامنه نسبت به توسعه رابط کاربری به همراه دارد.

بر پارادایم برنامه نویسی تابعی تسلط داشته باشید تا بتوانید مطالبی را که آموخته اید به طور هوشمندانه در حوزه های دیگر به کار ببرید. (برای نمونه) سیستم های شی شما با شفافیت ارجاعی هماهنگ می شوند و به همتاهای تابعی خود بسیار نزدیک تر از آن چیزی هستند که فکر می کنید. در واقع، برخی حتی ادعا می کنند که برنامه نویسی تابعی و شی­گرایی صرفاً بازتابی از یکدیگر هستند، نوعی از یین و یانگ محاسباتی.

**از خودتان بپرسید «کاربر چه کار می­کند؟» (شما کاربر نیستید)**

*گیلز کولبرن*

همه ما دوست داریم تصور کنیم که دیگران مانند ما فکر می کنند؛ اما آن­ها مثل ما فکر نمی­کنند. روانشناسان این موضوع را *تعصب اجماع کاذب* می نامند. وقتی مردم متفاوت از ما فکر یا عمل می‌کنند، احتمالاً (به طور ناخودآگاه) به نوعی به آنها برچسب معیوب بودن می‌زنیم.

این تعصب توضیح می دهد که چرا برنامه نویسان نمی­توانند خودشان را جای کاربران قرار دهند. کاربران مانند برنامه نویسان فکر نمی کنند. اولین تفاوت این است که آن­ها زمان بسیار کمتری را صرف استفاده از کامپیوتر می کنند. کاربران نه می دانند و نه اهمیت می دهند که یک کامپیوتر چگونه کار می کند. این بدان معناست که نمی توانند از هیچ یک از تکنیک های حل مسئله­ای استفاده کنند که برای برنامه نویسان آشناست. آن­ها الگوها و نشانه هایی را که برنامه نویسان برای کار با یک رابط استفاده می کنند، تشخیص نمی دهند.

بهترین راه برای فهمیدن اینکه کاربر چگونه فکر می کند، تماشای آن است. از یک کاربر بخواهید با استفاده از یک نرم افزار مشابه با آنچه شما در حال توسعه هستید، یک کار را تکمیل کند. اطمینان حاصل کنید که یک تسک واقعی است: «افزودن ستونی از اعداد» تسک خوبی است. «هزینه های ماه گذشته خود را محاسبه کنید» تسک بهتری است. از تسک­های ریز و با جزئیات خودداری کنید، مثلا «آیا می‌توانید این سلول‌های spreadsheet را انتخاب کنید و فرمول SUM را در زیر وارد کنید؟»— سرنخ بزرگی در این سؤال وجود دارد. کاربر را وادار کنید تا در مورد پیشرفت خود صحبت کند. حرفش را قطع نکنید. سعی نکنید کمک کنید. مدام از خود بپرسید: «چرا این کار را می کند؟» و «چرا این کار را نمی کند؟»

اولین چیزی که متوجه خواهید شد این است که کاربران مجموعه­ای از کارها را به طور مشابه انجام می دهند. آنها سعی می کنند تسک­ها را به همان ترتیب انجام دهند - و در موقعیت های مشابه اشتباهات مشابهی را انجام می دهند. شما باید حول آن رفتار اصلی برنامه را طراحی کنید. این کار با جلسات طراحی که تمایل دارند وقتی کسی می گوید، «اگر کاربر بخواهد فلان کار را انجام دهد چه؟» حرف آن را گوش دهند متفاوت است؛ این تفکر منجر به پیچیدگی ویژگی ها و سردرگمی در مورد خواسته­های کاربران می شود. تماشای کاربران این سردرگمی را از بین می برد.

خواهید دید کاربران گیر خواهند افتاد. وقتی کسی گیر می کند به اطراف نگاه می کند. وقتی کاربران گیر می کنند، تمرکز خود را محدود می کنند. دیدن راه حل ها در جای دیگری روی صفحه برای آنها سخت تر می شود. اگر باید دستورالعمل یا متن راهنما داشته باشید، مطمئن شوید که آن را دقیقاً در کنار مناطق مشکل ساز خود قرار دهید. علت اینکه tooltip ها مفیدتر از منوهای راهنما هستند، تمرکز محدود کاربر است.

کاربران تمایل به درهم ریختگی دارند. آنها راهی را پیدا خواهند کرد که کار کند و به آن پایبند باشند، مهم نیست چقدر پیچیده باشد. بهتر است به جای دو یا سه میانبر یک راه واقعا واضح برای انجام کارها ارائه کنید.

همچنین متوجه خواهید شد که بین آنچه کاربران می گویند می خواهند و آنچه واقعا انجام می دهند، فاصله وجود دارد. این موضوعی نگران‌کننده است، زیرا روش معمولی برای جمع‌آوری نیازهای کاربر این است که از آنها بپرسید. به همین دلیل است که بهترین راه برای درک نیازهای کاربران تماشای آن­ها است. یک ساعت تماشای کاربران مفیدتر از صرف یک روز برای حدس زدن آنچه آنها می خواهند است.

**استاندارد کدنویسی خود را خودکار کنید**

*فیلیپ ون لاینن*

احتمالا شما هم در این موقعیت بوده­اید. در ابتدای یک پروژه، همه هدف­های خوب زیادی دارند - آنها را «تصمیمات پروژه جدید» بنامید. اغلب، بسیاری از این تصمیمات در داکیومنت­ها نوشته می شوند. موارد مربوط به کد به استاندارد کدنویسی پروژه ختم می شود. در طول جلسه آغازین، توسعه‌دهنده اصلی داکیومنت را بررسی می‌کند و در بهترین حالت، همه موافق هستند که تا حد ممکن آن­ها را دنبال کنند. با این حال، پس از شروع پروژه، این اهداف خوب، یکی یکی کنار گذاشته می شوند. هنگامی که پروژه در نهایت تحویل داده شد، کد به نظر درهم و برهم است، و به نظر می رسد هیچ کس نمی داند چگونه به این شکل درآمده است.

چه زمانی همه چیز خراب شد؟ احتمالاً از قبل از جلسه آغازین. برخی از اعضای پروژه توجه نکردند. دیگران موضوع را درک نکردند. بدتر از همه، برخی مخالف بودند و در حال برنامه ریزی برای قالب کردن استاندارد کدنویسی خود بودند. بالاخره عده‌ای به این نکته رسیدند و موافقت کردند، اما وقتی فشار پروژه خیلی زیاد شد، مجبور شدند چیزهایی را رها کنند. کدهایی که به خوبی قالب بندی شده اند، برای مشتری که خواهان عملکرد بیشتر است، فایده­ای ندارند. علاوه بر این، پیروی از یک استاندارد کد نویسی اگر خودکار نباشد، می تواند کار بسیار خسته کننده ای باشد. فقط سعی کنید یک کلاس نامرتب را دستی فاصله دهید تا خودتان متوجه شوید.

اما اگر چنین مشکلی است، چرا در وهله اول یک استاندارد کدنویسی می خواهیم؟ یکی از دلایل قالب بندی کد به روشی یکنواخت این است که هیچ کس نمی تواند فقط با قالب بندی به روش شخصی خود، «صاحب» یک قطعه کد شود. برای جلوگیری از برخی باگ­های رایج، ممکن است بخواهیم از استفاده توسعه دهندگان از آنتی پترن­های خاصی جلوگیری کنیم. در کل، یک استاندارد کدنویسی باید کار در پروژه را آسان‌تر کند و سرعت توسعه را از ابتدا تا انتها حفظ کند. بنابراین، همه باید در مورد استاندارد کدنویسی نیز به توافق برسند - اگر یک توسعه‌دهنده از سه فاصله برای تورفتگی کد استفاده کند و دیگری از چهار فاصله استفاده کند، این کار به آسان تر شدن یا سرعت پروژه کمکی نمی­کند.

ابزارهای زیادی وجود دارد که می توان از آنها برای تولید گزارش های کیفیت کد و مستندسازی و حفظ استاندارد کدگذاری استفاده کرد، اما این ابزارها راه حل کاملی نیستند، این کارها باید به صورت خودکار انجام و در صورت امکان اجرا شوند. در اینجا چند راه­حل آورده شده است:

* اطمینان حاصل کنید که قالب‌بندی کد بخشی از فرآیند ساخت است، به طوری که همه هر بار که کد را کامپایل می‌کنند، این قالب بندی را به‌طور خودکار اجرا کنند.
* از ابزارهای تجزیه و تحلیل کد استاتیک برای اسکن کد آنتی پترن­های ناخواسته استفاده کنید. اگر موردی پیدا شد، ساخت را متوقف کنید.
* یاد بگیرید که این ابزارها را طوری پیکربندی کنید که بتوانید آنتی پترن­های خاص پروژه خود را اسکن کنید.
* نه تنها محدوده عمل تست بلکه به طور خودکار نتایج را نیز بررسی کنید. اگر محدوده عمل تست خیلی کم است، دوباره ساخت را متوقف کنید.

سعی کنید برای هر چیزی که مهم می دانید این کار را انجام دهید. شما نمی توانید هر چیزی را که واقعاً به آن اهمیت می دهید خودکار کنید. در مورد مواردی که نمی‌توانید به‌طور خودکار ثابت یا اصلاح کنید، آنها را مجموعه‌ای از دستورالعمل‌های مکمل استاندارد کدنویسی خودکار در نظر بگیرید، اما بپذیرید که ممکن است شما و همکارانتان آن‌ها را با جدیت دنبال نکنید.

در نهایت، استاندارد کدگذاری باید پویا باشد نه ایستا. با پیشرفت پروژه، نیازهای پروژه تغییر می کنند و آنچه در ابتدا هوشمندانه به نظر می رسید، لزوماً چند ماه بعد هوشمندانه نخواهد بود.

**زیبایی در سادگی است**

*یورن اولمهیم*

یک نقل قول از افلاطون وجود دارد که به نظر من برای همه توسعه دهندگان نرم افزار دانستن و به خاطر سپردن آن خوب است:

*زیبایی سبک و هارمونی و ظرافت و ریتم خوب به سادگی بستگی دارد.*

در یک جمله، این خلاصه ارزش هایی است که ما به عنوان توسعه دهندگان نرم افزار باید آرزوی آنها را داشته باشیم.

تعدادی ویژگی وجود دارد که باید در کدهایمان برای آن­ها تلاش می کنیم:

* خوانایی
* قابلیت نگهداری
* سرعت توسعه
* کیفیت دست­نیافتنی زیبایی

افلاطون به ما می گوید که عامل وجود همه این کیفیت ها سادگی است.

کد زیبا چیست؟ این سوال کاملا سلیقه­ای است. همانطور که درک ما از هر چیزی به پیشینه ما بستگی دارد، درک زیبایی نیز به شدت به پیشینه فردی بستگی دارد. افرادی که در هنر تحصیل کرده اند، درک متفاوتی (یا حداقل دیدگاه متفاوتی) از زیبایی نسبت به افراد تحصیل کرده در علوم دیگر دارند. رشته‌های هنر تمایل دارند با مقایسه نرم‌افزار با آثار هنری به زیبایی در نرم‌افزار نزدیک شوند، در حالی که رشته‌های علوم تمایل دارند در مورد تقارن و نسبت طلایی صحبت کنند و سعی می‌کنند مسائل را به فرمول تبدیل کنند. در تجربه من، سادگی اساس اکثر استدلال های هر دو طرف است.

به سورس کدی که مطالعه کرده اید فکر کنید. اگر وقت خود را صرف مطالعه کد دیگران نکرده اید، همین الان کد اپن سورسی برای مطالعه پیدا کنید. به دنبال کدی به زبان انتخابی خود باشید که توسط متخصصین مشهور و شناخته شده نوشته شده است.

برگشتید؟ خب، کجا بودیم؟ بله… آن کدی را پیدا کردم که با من هماهنگ است و آن را زیبا می دانم و دارای تعدادی ویژگی مشترک است. مهمترین این ویژگی­ها سادگی است. متوجه شدم که مهم نیست کل برنامه یا سیستم چقدر پیچیده باشد، بخش های جداگانه باید ساده نگه داشته شوند: اشیاء ساده با یک مسئولیت واحد حاوی متدهای مشابه ساده و متمرکز با نام های توصیفی. برخی افراد فکر می‌کنند که ایده داشتن متدهای کوتاه 5 تا 10 خطی بسیار افراطی است، و برخی از زبان‌ها انجام آن را بسیار سخت می‌کنند، اما با این حال من فکر می‌کنم که چنین اختصاری هدف مطلوبی است.

نکته نهایی این است که کد زیبا همان کد ساده است. هر بخش جداگانه با مسئولیت های ساده و روابط ساده با سایر بخش های سیستم ساده نگه داشته می شود. این راهی است که ما می‌توانیم سیستم‌های خود را در طول زمان، با کدهای تمیز، ساده و قابل آزمایش، حفظ کنیم و از سرعت بالای توسعه در طول عمر سیستم اطمینان حاصل کنیم.

زیبایی از سادگی زاده می شود و در سادگی یافت می شود.

**قبل از اینکه ریفکتور کنید**

*راجیت آتاپاتو*

از یک جایی به بعد، هر برنامه نویسی باید کدهای موجود را ریفکتور کند. اما قبل از انجام این کار، لطفاً به موارد زیر فکر کنید، این کار می‌تواند باعث صرفه‌جویی در وقت (و رنج) شما و دیگران شود:

* بهترین رویکرد برای ریفکتور کردن با بررسی کدبیس موجود و تست­های نوشته‌شده برای آن کد شروع می‌شود. این رویکرد به شما کمک می کند تا نقاط قوت و ضعف کد را همانطور که در حال حاضر وجود دارد درک کنید، بنابراین می توانید اطمینان حاصل کنید که نقاط قوت را حفظ کرده و از اشتباهات جلوگیری می کنید. همه ما فکر می‌کنیم که می‌توانیم بهتر از سیستم موجود انجام دهیم... تا زمانی که به چیزی بهتر یا حتی بدتر از تجسم قبلی دست یابیم، زیرا نتوانستیم از اشتباهات سیستم موجود عبرت بگیریم.
* از وسوسه بازنویسی همه چیز اجتناب کنید. بهترین کار این است که تا حد امکان از کدها استفاده مجدد کنید. مهم نیست که کد چقدر زشت است، قبلاً تست شده، بررسی شده است یا خیر. دور انداختن کدهای قدیمی - به خصوص اگر در مرحله تولید باشد - به این معنی است که ماه ها (یا سال ها) کد تست شده و سخت شده در نبرد را دور می اندازید که ممکن است راه حل های خاص و رفع اشکالاتی داشته باشد که از آنها اطلاعی ندارید. اگر این را در نظر نگیرید، ممکن است کد جدیدی که می نویسید همان باگ های مرموزی را نشان دهد که در کد قدیمی رفع شده بود. این کار در طی سال­ها باعث می­شود زمان، تلاش و دانش به دست آمده شما هدر برود.
* بسیاری از تغییرات تدریجی بهتر از یک تغییر عظیم هستند. تغییرات تدریجی به شما این امکان را می‌دهند تا از طریق بازخورد، مانند تست­ها، تأثیر روی سیستم را آسان‌تر اندازه‌گیری کنید. بعد از ایجاد تغییر، دیدن شکست­های متعدد در تست جالب نیست. این شکست­ها می توانند منجر به ناامیدی و فشار روحی و در نتیجه تصمیمات نامناسب شوند. یکی یکی روبرو شدن با چند شکست در تست آسان تر است و منجر به رویکردی قابل کنترل تر می شود.
* پس از هر نسخه، مهم است که اطمینان حاصل شود که تست های موجود موفق می شوند. اگر تست­های موجود برای پوشش تغییراتی که ایجاد کرده‌اید کافی نیست، تست­های جدیدی اضافه کنید. تست ها را بدون توجه به کد قدیمی دور نریزید. در ظاهر، برخی از این تست‌ها ممکن است برای طراحی جدید شما قابل اجرا نباشند، اما ارزشش را دارد که دلایل اضافه شدن این تست­ها را به صورت دقیق بررسی کنید.
* ترجیحات شخصی و نفس شما نباید دلیل ریفکتور کردن باشند. اگر چیزی خراب نشده است، چرا آن را تعمیر کنید؟ اینکه سبک یا ساختار کد با ترجیحات شخصی شما مطابقت ندارد دلیل معتبری برای ریفکتور کردن نیست. اینکه فکر کنید می توانید کار بهتری نسبت به برنامه نویس قبلی انجام دهید هم دلیل موجهی نیست.
* فناوری جدید نیز دلیل غیرقابل­قبولی برای ریفکتور کردن است. یکی از بدترین دلایل برای ریفکتور این است که کد فعلی با فناوری­های جدید امروزی فاصله زیادی دارد، و ما معتقدیم که یک زبان یا فریمورک جدید می تواند کارها را بسیار زیباتر انجام دهد. اگر تجزیه و تحلیل هزینه و فایده ثابت نکند که یک زبان یا فریمورک جدید به بهبودهای قابل توجهی در عملکرد، قابلیت نگهداری یا بهره وری منجر می شود، بهتر است آن را همانطور که هست رها کنید.
* به یاد داشته باشید که انسان ها اشتباه می کنند. ریفکتور کردن همیشه تضمین نمی‌کند که کد جدید بهتر یا حتی به خوبی کد قبلی باشد.

**مراقب استفاده مجدد باشید**

*اودی دهان*

اولین پروژه من در شرکت بود، تازه مدرکم را تمام کرده بودم و مشتاق بودم خودم را ثابت کنم و هر روز تا دیروقت می‌ماندم و کدهای موجود را مرور می‌کردم. زمانی که روی اولین فیچر خود کار می‌کردم، مراقبت بیشتری انجام دادم تا همه چیزهایی را که یاد گرفته‌ام در جای خود قرار دهم – کامنت گذاشتن، ورود به سیستم، بیرون کشیدن کد اشتراک‌گذاری شده در کتابخانه‌ها. در مرور کدی که احساس می‌کردم برای آن بسیار آماده بودم، حقیقت تلخی وجود داشت - استفاده مجدد از آن درست نبود!

چگونه می تواند اینگونه باشد؟ در زمان تحصیل در کالج، استفاده مجدد به عنوان مظهر مهندسی نرم افزار با کیفیت مطرح بود؛ تمام مقاله‌هایی که خوانده بودم، کتاب‌های درسی، متخصصان نرم‌افزار باتجربه‌ای که به من آموزش دادند، یعنی این تفکر اشتباه بود؟

معلوم شد که چیز مهمی را از دست داده ام: کانتکست.

این واقعیت که دو بخش کاملاً متفاوت از سیستم برخی از منطق را به یک روش اجرا می‌کردند، کمتر از آن چیزی که فکر می‌کردم اتفاق می افتاد. تا زمانی که آن کتابخانه های کد مشترک را بیرون نیاوردم، این بخش ها به یکدیگر وابسته نبودند. هر کدام می توانستند به طور مستقل تکامل یابند. هر کدام می توانستند منطق خود را متناسب با نیازهای محیط تجاری در حال تغییر سیستم عوض کنند. آن چهار خط کد مشابه، تصادفی بودند.

کتابخانه‌های کد اشتراکی که من ایجاد کردم، بند کفش هر پا را به پای دیگر می‌بست. مراحل یک دامنه تجاری را نمی توان بدون همگام سازی با دیگری انجام داد. هزینه های تعمیر و نگهداری در آن توابع مستقل قبلا ناچیز بود، اما کتابخانه رایج نیاز به تست بزرگتری داشت.

در حالی که تعداد مطلق خطوط کد را در سیستم کاهش داده بودم، تعداد وابستگی ها را افزایش داده بودم. کانتکست این وابستگی‌ها بسیار مهم است - اگر بومی‌سازی می‌شدند، ممکن بود اشتراک‌گذاری منطقی باشد و ارزش مثبتی داشت. حتی اگر خود کد خوب به نظر برسد وقتی این وابستگی‌ها کنترل نشوند، نگرانی های بزرگ تری در سیستم به وجود می آورند.

این اشتباهات از این جهت خطرناک هستند که در مجموع ایده خوبی به نظر می رسند. هنگامی که این تکنیک ها در کانتکست مناسب اعمال شوند، ارزشمند هستند. در کانتکست اشتباه، به جای مفید بودن، هزینه را افزایش می دهند. این روزها وقتی وارد یک کدبیس موجود می‌شوم بدون اینکه اطلاعاتی در مورد محل استفاده از بخش‌های مختلف نداشته باشم، خیلی مراقب چیزهایی هستم که دوباره استفاده می‌شوند.

مراقب استفاده مجدد باشید. کانتکست خود را بررسی کنید. سپس ادامه دهید.

**قانون بوی اسکات یا قانون پیشاهنگ**

*رابرت سی مارتین (آنکل باب)*

پیشاهنگان قانونی دارند: «همیشه کمپ را تمیزتر از زمانی که وارد شدید، ترک کنید.» اگر دیدید زمین نامرتب است، بدون توجه به اینکه چه کسی ممکن است این نامرتبی را ایجاد کرده باشد، آن را تمیز می کنید. شما عمداً محیط را برای گروه بعدی کمپ­نشینان بهبود می بخشید. (در واقع، شکل اصلی این قانون که توسط رابرت استفنسون اسمیت بادن پاول، پدر پیشاهنگی نوشته شده بود، این بود: «سعی کن و این دنیا را کمی بهتر از آنچه که پیدا کردی ترک کن»).

اگر از قانون مشابهی در کدهای خود پیروی کنیم، چه اتفاقی می­افتد: «همیشه یک ماژول را تمیزتر از زمانی که آن را دیدید، بررسی کنید»؟ صرف نظر از اینکه نویسنده اصلی چه کسی بوده است، چه می‌شود اگر ما همیشه، هر چقدر هم کوچک، تلاشی برای بهبود ماژول انجام دهیم؟ نتیجه چه خواهد بود؟

من فکر می کنم اگر همه ما از این قانون ساده پیروی کنیم، شاهد پایان نابودی پیاپی سیستم های نرم افزاری خود خواهیم بود. در عوض، سیستم های ما به تدریج با توسعه بهتر و بهتر می شوند. همچنین شاهد تیم هایی خواهیم بود که از سیستم به عنوان یک کل مراقبت می کنند، نه اینکه هر فرد فقط به بخش کوچک خود اهمیت دهد.

فکر نمی‌کنم این قانون خواسته­ی خیلی زیادی باشد. لازم نیست قبل از بررسی هر ماژول، آن را اصلاح کنید. فقط باید آن را به ماژولی کمی بهتر از زمانی که بررسی­اش می‌کنید، تبدیل کنید. البته این بدان معناست که هر کدی که به یک ماژول اضافه می کنید باید تمیز باشد. همچنین به این معنی است که قبل از بررسی مجدد ماژول، حداقل یک بخش دیگر را تمیز کنید. صرفا ممکن است نام متغیری را اصلاح کنید، یا تابعی طولانی را به دو تابع کوچکتر تقسیم کنید. ممکن است یک وابستگی دایره ای را بشکنید یا اینترفیسی برای جدا کردن خط مشی از جزئیات اضافه کنید.

مانند اصول اولیه زندگی به نظر می رسد - مثل شستن دست ها قبل از غذا خوردن، یا انداختن زباله ها در سطل به جای زمین. در واقع، رها کردن آشفتگی­ها در کد باید از نظر اجتماعی به اندازه ریختن زباله غیرقابل قبول باشد؛ اصلا نباید انجام شود.

اما از همه این­ها گذشته، مراقبت از کد خودمان یک چیز است، مراقبت از کد تیم چیز دیگری. تیم ها به یکدیگر کمک می کنند و کدهای همدیگر را تمیز میکنند. آنها از قانون پیشاهنگی پیروی می کنند زیرا این کار نه فقط برای خودشان بلکه برای همه خوب است.

**قبل از اینکه دنبال پیدا کردن مقصر باشید، ابتدا کد خودتان را بررسی کنید**

*آلن کِلی*

توسعه دهندگان – یعنی همه ما! - اغلب نمی‌توانیم باور کنیم مشکل از کد خودمان است. بسیار غیرممکن است که کامپایلر خراب شده باشد.

با این حال، در واقعیت، بسیار نادر است که کد توسط باگی در کامپایلر، مفسر، سیستم عامل، سرور برنامه، دیتابیس، مموری منیجر یا هر بخش دیگر از نرم افزار سیستم خراب شود. بله، این اشکالات وجود دارند، اما بسیار کمتر از چیزی هستند که بخواهیم به آن­ها تکیه کنیم.

تنها یک بار مشکلی واقعی با یک باگ کامپایلر داشتم که یک متغیر حلقه را بهینه می‌کرد، من تصور می‌کردم کامپایلر یا سیستم‌عامل باید خیلی بیشتر از اینها باگ داشته باشد. مدت زیادی از وقت، زمان پشتیبانی و مدیریت خود را در این فرآیند تلف کرده‌ام، اما هر بار که بعد از معلوم می‌شود که اشتباه از من بوده، کمی احساس حماقت می‌کنم.

با فرض اینکه ابزارها به طور گسترده مورد استفاده قرار می گیرند، توسعه پیدا می کنند و در زیرساخت های تکنولوژی مختلف به کار می روند، دلیلی برای شک در کیفیت آن ها وجود ندارد. البته، اگر این ابزار نسخه اولیه هستند، تنها توسط افراد معدودی در سراسر جهان استفاده می­شوند، به ندرت دانلود شده­اند، اوپن سورس هستند، یا نسخه 0.1 هستند، میتوان به نرمافزار شک کرد. (به همین ترتیب، نسخه آلفا نرم افزار تجاری ممکن است خیلی قابل اعتماد نباشد.)

با توجه به نادر بودن باگ های کامپایلر، بهتر است زمان و انرژی خود را برای یافتن خطای کد خود صرف کنید تا ثابت کنید که کامپایلر اشتباه کرده است. تمام توصیه‌های رایج دیباگ اعمال می‌شود، بنابراین مشکل را شناسایی کنید، ارتباطات را قطع کنید، و آن را با تست­های فراوان احاطه کنید. قراردادهای فراخوانی، کتابخانه های مشترک، و شماره نسخه را بررسی کنید. مشکل را برای کسی دیگر توضیح دهید؛ مراقب خرابی استک و عدم تطابق نوع متغیر باشید. و کد را در دستگاه ها و پیکربندی های مختلف ساخت مانند دیباگ و ریلیز امتحان کنید.

فرضیات خود و دیگران را زیر سوال ببرید. ممکن است فرضیات متفاوتی در ابزارهای عرضهکنندگان مختلف وجود داشته باشد - همچنین ممکن است ابزارهای متفاوتی از یک عرضهکننده در دسترس باشد.

وقتی شخص دیگری مشکلی را گزارش می‌کند که نمی‌توانید آن را شبیه سازی کنید، بروید و ببینید که او با آن مشکل چگونه رفتار می­کند. آنها ممکن است کاری را انجام دهند که شما هرگز فکرش را نکرده اید یا کاری را به ترتیب دیگری انجام دهند.

قانون شخصی من این است که اگر باگی داشته باشم که نمی توانم آن را پیدا کنم و فکر می کنم که اشکال از کامپایلر است، وقت آن است که به دنبال خرابی استک بگردم. این امر به ویژه در صورتی صدق می­کند که افزودن کد ردیابی باعث جابجایی مشکل شود.

مشکلات مالتی ترد یا چند رشته ای منبع دیگری از باگها هستند که هم برای توسعه دهنگان و هم برای دستگاه ها دردسرساز هستند. همه توصیه‌ها برای استفاده از کد ساده زمانی که یک سیستم چند رشته‌ای است، چند برابر می‌شود. برای یافتن چنین اشکالاتی با هر گونه سازگاری نمی توان به دیباگ کردن و یونیت تست ها اعتماد کرد، بنابراین سادگی طراحی بسیار مهم است.

بنابراین قبل از اینکه عجله کنید تا کامپایلر را سرزنش کنید، توصیه شرلوک هلمز را به یاد بیاورید: «وقتی غیرممکن‌ها را حذف کردید، هر آنچه باقی می‌ماند، هر چقدر هم دور از ذهن باشد، حقیقت است» و آن را به جای توصیه *دِرک جنتلی* انتخاب کنید، «وقتی غیرممکن‌ها را حذف کردید. هرچه باقی بماند، هر چقدر هم دور از ذهن باشد، حقیقت است.»

**ابزارهای خود را با دقت انتخاب کنید**

*جیووانی آسپرونی*

برنامه های مدرن به ندرت از صفر ساخته می شوند. آنها با استفاده از ابزارهای موجود – کامپوننت ها، کتابخانه‌ها و فریمورک ها – به دلایل مختلفی ساخته می‌شوند:

* اپلیکیشن ها از نظر اندازه و پیچیدگی رشد می کنند، در حالی که زمان برای توسعه آنها کوتاه تر می شود. اگر توسعه دهندگان بتوانند روی نوشتن کد business-domain بیشتر و کد زیرساخت کمتر تمرکز کنند، از زمان و هوش توسعه دهندگان بهتر استفاده می شود.
* کامپوننت‌ها و فریم‌ورک‌های پرکاربرد احتمالاً دارای باگهای کمتری نسبت به مواردی هستند که توسط خود توسعه دهنده ساخته شده‌اند.
* تعداد زیادی نرم افزار با کیفیت بالا به صورت رایگان در وب وجود دارد که به معنای هزینه های توسعه کمتر و احتمال بیشتر پیدا کردن توسعه دهندگان با علاقه و تخصص لازم است.
* تولید و نگهداری نرم‌افزار کاری است که به افراد زیادی نیاز دارد، بنابراین خرید نرم­افزار ممکن است ارزان‌تر از ساختن آن باشد.

با این حال، انتخاب ترکیبی مناسب از ابزارها برای برنامه شما می تواند کار دشواری باشد که نیاز به تفکر دارد. در واقع، هنگام انتخاب باید چند نکته را در نظر داشته باشید:

* ابزارهای مختلف در مورد کانتکست خود ممکن است بر فرضیات مختلفی متکی باشند - به عنوان مثال، زیرساخت محیط، کنترل مدل، دیتا مدل، پروتکل های ارتباطی، و غیره - که می تواند منجر به عدم تطابق معماری بین اپلیکیشن و ابزار شود. چنین عدم تطابقی منجر به هک و راه حل هایی می شود که کد را پیچیده تر از حد لازم می کنند.
* ابزارهای مختلف چرخه عمر متفاوتی دارند و ارتقاء یکی از آنها ممکن است به کاری بسیار دشوار و زمان بر تبدیل شود زیرا عملکرد جدید، تغییرات طراحی یا حتی رفع باگ ها ممکن است باعث ناسازگاری با ابزارهای دیگر شود. هرچه تعداد ابزارها بیشتر باشد، مشکل می تواند بدتر شود.
* برخی از ابزارها به پیکربندی بسیار کمی نیاز دارند، اغلب با استفاده از یک یا چند فایل XML، که می توانند خارج از کنترل خیلی سریع رشد کنند. ممکن است در نهایت به نظر برسد که انگار تمام برنامه با XML به اضافه چند خط کد عجیب با یک زبان برنامه نویسی نوشته شده است. پیچیدگی پیکربندی، نگهداری و گسترش برنامه را دشوار خواهد کرد.
* زمانی عرضه کننده دیگر نمی تواند ابزاری را گسترش دهد که کدی که به شدت به محصولات عرضه کننده خاص وابسته است، در نهایت توسط آن‌ محصولات در چندین مورد محدود شود: قابلیت نگهداری، عملکرد، توانایی تکامل و توسعه، هزینه و غیره.
* اگر قصد دارید از نرم افزار رایگان استفاده کنید، ممکن است متوجه شوید که در نهایت آنقدرها هم رایگان نیست. ممکن است نیاز به خرید پشتیبانی تجاری داشته باشید که لزوماً ارزان نخواهد بود.
* شرایط صدور مجوز، حتی برای نرم افزارهای رایگان هم مهم است. به عنوان مثال، در برخی از شرکت ها، استفاده از نرم افزار دارای مجوز تحت شرایط مجوز گنو به دلیل ماهیت ویروسی آن قابل قبول نیست - یعنی نرم افزار توسعه یافته توسط آن باید همراه با سورس کد توزیع شود.

استراتژی شخصی من برای کاهش این مشکلات این است که فقط با استفاده از ابزارهایی که کاملاً ضروری هستند شروع کوچکی داشته باشم. معمولاً تمرکز اولیه روی از بین بردن نیاز به برنامه‌نویسی (و مشکلات) زیرساخت‌های سطح پایین است، به عنوان مثال، با استفاده از برخی میان‌افزارها به جای استفاده از سوکت‌های خام برای برنامه‌های کاربردی توزیع شده و سپس در صورت نیاز استفاده از موارد بیشتر. من همچنین تمایل دارم ابزارهای خارجی را با استفاده از اینترفیس ها و لایه بندی از اشیاء دامنه کسب و کار خود جدا کنم تا در صورت لزوم با حداقل زحمت بتوانم ابزار را تغییر دهم. یک اثر جانبی مثبت این رویکرد این است که من معمولاً با برنامه­ای کوچکتر مواجه می شوم که از ابزارهای خارجی کمتری نسبت به پیش بینی اولیه استفاده می کند.

**کد به زبان دامنه**

*دَن نورث*

دو کدبیس را تصور کنید. در یکی، با این موارد مواجه می شوید:

if (portfolioIdsByTraderId.get(trader.getId())

.containsKey(portfolio.getId())) {...}

به این فکر می کنید که کاربرد این کد چیست. به نظر می رسد در حال گرفتن شناسه از یک شی trader و استفاده از آن برای به دست آوردن ظاهراً یک map از دو map دیگر است و سپس مشاهده این است که آیا شناسه دیگری از یک شیء پورتفولیو در نقشه داخلی وجود دارد یا خیر. مقداری دیگر فکر می کنید. شما به دنبال اعلان portfolioIdsByTraderId می گردید و این را کشف می کنید:

Map<int, Map<int, int>> portfolioIdsByTraderId;

به تدریج، متوجه می شوید که ممکن است ارتباطی با دسترسی یک trader به یک پورتفولیوی خاص داشته باشد. و البته هر زمان مهم باشد که آیا trader به یک پورتفولیو دسترسی دارد یا خیر، شما همان بخش lookup را پیدا خواهید کرد - یا به احتمال زیاد، یک قطعه کد مشابه اما به طور ماهرانهای متفاوت. در کدبیس دیگر، با این روبرو می شوید:

if (trader.canView(portfolio)) {...}

نیازی به فکر کردن نیست. لازم نیست بدانید که یک trader چگونه این چیزها را می داند. شاید یکی از این map های تودرتو جایی در داخل پنهان شده باشد. اما این کار trader است نه شما.

اکنون ترجیح می دهید روی کدام یک از آن کدبیسها کار کنید؟

روزی روزگاری، فقط ساختمان­های داده بسیار ابتدایی داشتیم: بیت ها و بایت ها و کاراکترها (واقعا فقط بایت ها، اما وانمود می کردیم که آنها حروف و علائم (نگارشی) هستند). اعداد بر پایه 10 کمی مشکلساز بودند، زیرا در باینری خیلی خوب کار نمی کنند، بنابراین چندین اندازه از انواع ممیز-شناور داشتیم. سپس آرایه ها و رشته ها (حقیقتا فقط آرایه های متفاوت) آمدند. سپس استک ها و queue ها و هش‌ها و لیست‌های پیوندی و جهشی و بسیاری دیگر از ساختمان­های داده هیجان‌انگیز داشتیم که در دنیای واقعی وجود ندارند. «علوم کامپیوتر» در مورد تلاش برای ارتباط بین دنیای واقعی و ساختمان­های داده محدود کننده ما بود. اساتید واقعی حتی می توانستند به یاد بیاورند که چگونه این کار را انجام داده اند.

سپس نوع­های تعریف شده توسط کاربر را دریافت کردیم! خب، این خبر جدیدی نیست، اما تا حدودی داستان را تغییر می دهد. اگر دامنه شما حاوی مفاهیمی مانند traderها و پورتفولیو است، می‌توانید آنها را با نوع­هایی به نام‌های مثلاً Trader و Portfolio مدل کنید. اما مهمتر از این، می توانید روابط بین آنها را با استفاده از اصطلاحات دامنه نیز مدل کنید.

اگر با استفاده از اصطلاحات دامنه کدنویسی نمی کنید، درک ضمنی (بخوانید: پنهان) ایجاد می کنید که این int در اینجا به معنای راهی برای شناسایی یک trader است، در حالی که int در آنجا به معنای راهی برای شناسایی یک پورتفولیو است. (بهتر است آنها را با هم قاطی نکنید!) و اگر مفهومی تجاری («بعضی از trader ها مجاز به مشاهده برخی از پورتفولیوها نیستند—غیرقانونی است») را با قطعه­ای الگوریتمی نشان می دهید—مثلاً یک رابطه وجودی در یک map از کلیدها—در ممیزی و انطباق هیچ کمکی نمی کنید.

برنامه نویس بعدی که می آید ممکن است به این نکته توجهی نداشته باشد، پس چرا آن را به صراحت بیان نکنیم؟ استفاده از یک کلید به عنوان lookup برای کلید دیگری که موجود بودن را بررسی می کند، چندان واضح نیست. چگونه قرار است کسی بفهمد که قوانین تجاری که از تعارض منافع جلوگیری می کنند آنجا پیاده سازی می شوند؟

واضح کردن مفاهیم دامنه در کد شما به این معنی است که سایر برنامه نویسان می توانند هدف کد را بسیار راحت تر درک کنند تا تلاش برای اصلاح الگوریتم به آنچه در مورد یک دامنه می دانند،. همچنین به این معنی است که وقتی مدل دامنه تکامل می‌یابد - که با افزایش درک شما از دامنه تکامل خواهد یافت- در موقعیت خوبی برای توسعه کد هستید. همراه با کپسوله سازی خوب، این احتمال وجود دارد که این قانون فقط در یک مکان وجود داشته باشد، و شما بتوانید آن را بدون اینکه کدهای وابسته عاقلانه تر باشند، تغییر دهید.

برنامه نویسی که چند ماه بعد سراغ کدتان می آید از شما تشکر خواهد کرد. آن برنامه نویسی ممکن است خود شما باشید.

**کد طراحی است**

*رایان براش*

تصور کنید فردا از خواب بیدار می­شوید و می­فهمید صنعت ساخت و ساز باعث پیشرفت قرن شده است. میلیون‌ها ربات ارزان قیمت و فوق‌العاده سریع می‌توانند از هوای رقیق، مواد مختلفی بسازند، هزینه برق تقریباً صفر دارند و می‌توانند خودشان را تعمیر کنند. و بهتر هم می‌شود: با توجه به طرحی واضح برای پروژه­ای ساختمانی، ربات‌ها می‌توانند آن را بدون دخالت انسان و با هزینه ناچیز بسازند.

می توان تاثیر آن بر صنعت ساخت و ساز را تصور کرد، اما در بالادست چه اتفاقی می افتد؟ اگر هزینه های ساخت و ساز ناچیز بود، رفتار معماران و طراحان چگونه تغییر می کرد؟ امروزه مدل های فیزیکی و کامپیوتری قبل از سرمایه گذاری در ساخت و ساز ساخته و به شدت آزمایش می شوند. اگر ساخت و ساز اساساً رایگان بود، زحمت می کشیدیم؟ اگر طرحی از بین برود، چیز مهمی نیست-فقط متوجه شوید که چه مشکلی رخ داده است و از ربات‌های جادویی ما بخواهید که یکی دیگر بسازند. پیامدهای دیگری نیز وجود دارد. با مدل‌های منسوخ، طرح‌های ناتمام با ایجاد و بهبود مکرر بر اساس برآورد هدف نهایی تکامل می‌یابند. یک ناظر معمولی ممکن است در تشخیص یک طرح ناتمام از یک محصول نهایی مشکل داشته باشد.

نمی­توانیم جدول زمانی را پیش­بینی کنیم. هزینه های ساخت و ساز آسان تر از هزینه های طراحی محاسبه می شود - ما هزینه برآورد نصب یک تیرچه و اینکه چند تیرچه نیاز داریم را می دانیم. همانطور که وظایف قابل پیش بینی به سمت صفر کاهش می یابند، پیش­بینی زمان طراحی سخت­تر می شود. نتایج سریعتر تولید می شوند، اما جدول زمانی قابل اطمینان از بین می رود.

البته فشارهای اقتصاد رقابتی همچنان وجود دارند. با حذف هزینه های ساخت و ساز، شرکتی که می تواند طراحی را سریع تکمیل کند، در بازار برتری پیدا می کند. شرکت های مهندسی به سمت طراحی سریع حرکت می کنند. به ناچار، شخصی که عمیقاً با طراحی آشنا نیست، نسخه­ای غیرمعتبر و مزیت عرضه زودهنگام را می بیند و می گوید: «به اندازه کافی خوب به نظر می رسد.»

برخی از پروژه های حیاتی سخت، دقیق تر خواهند بود؛ اما در بسیاری از موارد، مصرف کنندگان یاد می گیرند که از طراحی ناقص رنج ببرند. شرکت‌ها همیشه می‌توانند ربات‌های جادویی ما را بفرستند تا ساختمان‌ها و وسایل نقلیه خرابی را که می‌فروشند «تعمیر کنند». همه این‌ها به نتیجه‌گیری شگفت‌انگیز غیرمعمولی­ای اشاره می‌کنند: فرض ما تنها کاهش چشمگیر هزینه‌های ساخت و ساز بود که در نتیجه کیفیت بدتر شد.

نباید تعجب کنیم که داستان قبلی درمورد نرم افزار اتفاق افتاده است. اگر بپذیریم که کد طراحی است یعنی فرآیندی خلاقانه نه مکانیکی، بحران نرم افزار توضیح داده می شود. ما اکنون با بحران طراحی روبرو هستیم: تقاضا برای طرح های با کیفیت و معتبر، فراتر از ظرفیت ما برای ایجاد آنها است. فشار شدیدی برای استفاده از طراحی ناقص وجود دارد.

خوشبختانه، این مدل، سرنخ هایی برای بهتر شدن ما نیز ارائه می دهد. شبیه سازی های فیزیکی معادل تست خودکار است. طراحی نرم‌افزار کامل نمی‌شود تا زمانی که با مجموعه عظیمی از تست ها تأیید شود. زبان های بهبود یافته و شیوه های طراحی امیدبخش هستند. در نهایت، واقعیتی اجتناب‌ناپذیر وجود دارد: طرح‌های عالی توسط طراحان بزرگی تولید می‌شوند که خود را وقف تسلط بر هنر خود می‌کنند. در این مورد تفاوتی بین کد و طرح وجود ندارد.

**چیدمان کد مهم است**

*استیو فِریمَن*

چندین سال پیش، روی یک سیستم Cobol کار کردم که در آن کارکنان اجازه نداشتند تورفتگی را تغییر دهند، مگر اینکه از قبل دلیلی برای تغییر کد داشته باشند، زیرا یک بار شخصی از دستش در رفت و با قرار دادن یک خط در یکی از ستون های ویژه در ابتدای یک خط، چیزی را خراب کرد. این مشکل حتی اگر چیدمان گمراه‌کننده بود نیز رخ می داد، که گاهی اوقات گمراه کننده هم بود، بنابراین مجبور بودیم کد را با دقت بخوانیم زیرا نمی‌توانستیم به آن اعتماد کنیم. این سیاست باید هزینه گزافی را برای برنامه نویسان داشته باشد.

تحقیقاتی وجود دارد که نشان می‌دهد همه ما بیشتر از زمان برنامه‌نویسی خود را صرف پیمایش و خواندن کد می‌کنیم – پیدا کردن اینکه کجا را تغییر دهیم - تا در واقع تایپ کردن، بنابراین این همان چیزی است که می‌خواهیم آن را بهینه کنیم. در ادامه سه روش برای بهینه سازی گفته می شود:

اسکن آسان

مردم در تطبیق الگوهای بصری واقعاً خوب هستند، بنابراین می‌توانم به خودم کمک کنم و هر چیزی را که مستقیماً به دامنه مرتبط نیست - همه «پیچیدگی های تصادفی» که با اکثر زبان های تجاری ارائه می شود - با استانداردسازی آن، در پس زمینه محو کنم. اگر کدی که رفتار یکسانی دارد یکسان به نظر می رسد، سیستم ادراکی من به من کمک می کند تا تفاوت ها را تشخیص دهم. به همین دلیل است که قراردادهایی را در مورد نحوه چیدمان بخش‌های یک کلاس در یک واحد کامپایل نیز رعایت می‌کنم: کانستنت ها، فیلدها، متدهای عمومی، متدهای خصوصی.

چیدمان رسا

همه ما یاد گرفته‌ایم که برای یافتن نام‌های مناسب وقت بگذاریم تا کد ما صریحا بیان کند که چه کاری انجام می‌دهد، نه اینکه فقط مراحل را فهرست کند – درست است؟ طرح کد نیز بخشی از این صریح و رسا بودن است. اولین برداشت این است که تیم بر روی یک فرمتر خودکار برای اصول اولیه به توافق برسد، سپس ممکن است در حین کدنویسی تنظیمات را دستی انجام دهند. تا زمانی که اختلاف نظر خاصی وجود نداشته باشد، افراد تیم به سرعت کارشان را به صورت دستی پیش می­برند. یک فرمتر نمی‌تواند مقاصد من را بفهمد و برای من این مهم‌تر است که سر خط بعدی رفتن ها و گروه‌بندی‌ها منعکس کننده هدف کد باشد، نه فقط سینتکس زبان.

فرمت فشرده

هرچه بدون شکستن کانتکست با پیمایش یا جابجایی فایل‌ها، بتوانم کدهای بیشتری روی صفحه قرار دهم، بیشتر می‌توانم ببینم، یعنی می‌توانم حالات کمتری را در ذهنم نگه دارم. کامنت های طولانی و تعداد فضاهای خالی زیاد برای نام های هشت کاراکتری و پرینترهای خط، منطقی بود، اما اکنون از یک IDE استفاده می کنم که سینتکس و کراس لینک­ها را رنگ می کند. پیکسل ها عامل محدودکننده من هستند، بنابراین می خواهم همه­ی چیزها در درک من از کد مشارکت کنند. می‌خواهم چیدمان در درک کد به من کمک کند، اما فقط در درک کد نه چیزی بیشتر.

یکی از دوستانم که برنامه­نویس نیست یک بار گفت کد شبیه شعر است. من این احساس را از کدهای بسیار خوب دریافت می‌کنم - اینکه همه چیز در متن یک هدف دارد و برای کمک به درک ایده است. متأسفانه، کدنویسی همان تصویر عاشقانه سرودن شعر را ندارد.

**بررسی کد**

*ماتیاس کارلسون*

شما باید کد را بررسی کنید. چرا؟ زیرا کیفیت کد را افزایش و میزان عیب و نقص آن را کاهش می دهد. اما نه لزوماً به دلایلی که ممکن است به ذهنتان خطور کند.

بسیاری از برنامه نویسان از آنجا که ممکن است قبلاً تجربیات بدی درمورد بررسی کد داشته باشند، تمایلی از خود نشان نمی دهند. سازمان‌هایی را دیده‌ام که نیاز دارند همه کدها قبل از استقرار در تولید، بازبینی رسمی داشته باشند. اغلب، معمار یا توسعه‌دهنده اصلی این بررسی را انجام می‌دهد، عملی که می‌توان آن را به‌عنوان «بررسی همه چیز توسط معمار» توصیف کرد. این موضوع در کتابچه راهنمای فرآیند توسعه نرم افزار شرکت بیان شده است، بنابراین برنامه نویسان باید آن را رعایت کنند.

ممکن است برخی از سازمان ها باشند که به چنین فرآیند سفت و سخت و رسمی نیاز داشته باشند، اما بیشتر سازمان ها نیاز ندارند. در اکثر سازمان ها، چنین رویکردی نتیجه عکس دارد. ارزیابان می توانند احساس کنند که توسط هیئت عفو مورد قضاوت قرار می گیرند. ارزیابان هم به زمان لازم برای خواندن کد و هم به زمان لازم برای به روز بودن در مورد تمام جزئیات سیستم نیاز دارند.

به جای تصحیح ساده اشتباهات کد، هدف از بررسی کد باید به اشتراک گذاری دانش و ایجاد دستورالعمل های کدگذاری مشترک باشد. اشتراک گذاری کد با سایر برنامه نویسان، کدنویسی دسته جمعی را امکان­پذیر می­کند. به یکی از اعضای تیم به طور تصادفی اجازه دهید با بقیه اعضای تیم کد را بررسی کند. به جای جستجوی خطاها، باید تلاش کنید کد را برای یادگیری و درک آن مرور کنید.

در هنگام بررسی کدها آرام باشید. اطمینان حاصل کنید که نظرات سازنده هستند، نه انتقادی و تند. نقش‌های مختلف را برای جلسه بررسی معرفی کنید تا از تأثیرگذاری ارشدیت سازمانی بین اعضای تیم در بررسی کد جلوگیری کنید. نمونه‌هایی از نقش‌ها می‌تواند شامل تمرکز یک بازبین بر مستندات، دیگری بر روی استثناها، و سومی برای بررسی عملکرد باشد. این رویکرد کمک می کند تا بار بررسی بین اعضای تیم تقسیم شود.

هر هفته یک روز به طور منظم مرور کد داشته باشید. چند ساعتی را صرف یک جلسه نقد و بررسی کنید. هر جلسه بازبینی را با یک الگوی ساده چرخشی بچرخانید. به یاد داشته باشید که در هر جلسه نقش بین اعضای تیم را نیز تغییر دهید. افراد تازه کار را در بررسی کد مشارکت دهید؛ ممکن است بی تجربه باشند، اما دانش جدید دانشگاهی آنها می تواند دیدگاه متفاوتی را ارائه دهد. متخصصان را برای تجربه و دانششان مشارکت دهید؛ آنها کدهای مستعد خطا را سریعتر و با دقت بیشتری شناسایی می کنند. در صورتی که تیم، قوانین کدنویسی داشته باشد که توسط ابزارها بررسی می شود، بررسی کدها راحت تر انجام می شود. به این ترتیب، قالب بندی کد هرگز در جلسه بررسی کد مورد بحث قرار نخواهد گرفت.

شاید مهمترین عامل موفقیت، سرگرم کننده کردن مرور و بررسی کد باشد. بررسی کدها به افرادی که بررسی می کنند، مربوط است؛ بنابراین، اگر جلسه بررسی عداب آور یا کسل کننده باشد، انگیزه داشتن دشوار خواهد بود. آن را به یک بررسی کد غیررسمی تبدیل کنید که هدف اصلی آن اشتراک گذاری دانش بین اعضای تیم است. نظرات طعنه آمیز را کنار بگذارید و به جای آن با خودتان عصرانه یا ناهار بیاورید.

**کدنویسی با دلیل**

*یِچیل کیمچی*

تلاش برای استدلال دستی در مورد صحت نرم‌افزار منجر به درستی­یابی صوری می‌شود که طولانی‌تر از کد است و احتمالاً حاوی خطا است. برای این کار ابزارهای خودکار ارجحیت دارند اما استفاده از آن ها همیشه ممکن نیست. آنچه در ادامه گفته می شود مسیری میانه را توصیف می کند: استدلال نیمه صوری در مورد صحت نرم­افزار.

رویکرد اساسی این است که تمام کدهای مورد بررسی را به بخش‌های کوچک­تر تقسیم کنیم - از یک خط واحد، مانند فراخوانی تابع، تا بلاک‌های کد کمتر از 10 خط – و درمورد صحت آن ها بحث کنیم. استدلال ها فقط باید به اندازه کافی قوی باشند تا همکار ساز مخالف­زن شما را متقاعد کنند.

یک بخش باید به گونه ای انتخاب شود که در هر اندپوینتی، استیت برنامه (یعنی شمارنده برنامه و مقادیر تمام اشیاء «موجود») یک پراپرتی که به راحتی توصیف شده را برآورده کند، به طوری که عملکرد آن بخش (تغییر استیت) به آسانی به عنوان یک تسک توصیف می شود. این دستورالعمل ها استدلال را ساده تر می کنند. چنین پراپرتی­های اندپوینتی، مفاهیمی مانند پیش‌شرط‌ها و پس‌شرط‌ها را برای توابع، و ناورداها را برای حلقه‌ها و کلاس‌ها (با توجه به نمونه‌های آنها) تعمیم می‌دهند. تلاش برای مستقل بودن بخش‌ها از یکدیگر تا حد امکان، استدلال را ساده می‌کند و زمانی که این بخش‌ها باید اصلاح شوند، ضروری است.

بسیاری از شیوه‌های کدنویسی که به خوبی شناخته شده‌اند (اگرچه شاید کمتر دنبال شوند) و «خوب» در نظر گرفته می‌شوند، استدلال را آسان‌تر می‌کنند. از این رو، فقط با قصد استدلال در مورد کد خود، در حال حاضر می توانید به سمت سبک و ساختار بهتری حرکت کنید. عجیب نیست که اکثر این روش ها را می توان توسط تحلیلگرهای کد استاتیک بررسی کرد:

* از دستورات goto استفاده نکنید، زیرا بخش های دور از هم را به شدت به یکدیگر وابسته می کنند.
* از متغیرهای گلوبال قابل تغییر استفاده نکنید، زیرا این متغیرها همه بخش هایی را که از آنها استفاده می کنند، وابسته می کنند.
* هر متغیر باید کمترین دامنه ممکن را داشته باشد. به عنوان مثال، یک شی لوکال را می توان درست قبل از اولین استفاده آن اعلام کرد.
* هر زمان که نیاز است، اشیا را imutable یا تغییرناپذیر کنید.
* کد را با استفاده از فاصله افقی و عمودی خوانا کنید - به عنوان مثال، تراز کردن ساختارهای مرتبط و استفاده از یک خط خالی برای جدا کردن دو بخش.
* با انتخاب نام‌های توصیفی (اما نسبتاً کوتاه) برای اشیا، تایپ­ها، توابع و ... ، کد را مستندسازی کنید.
* اگر به بخشی تودرتو نیاز دارید، آن را به یک تابع تبدیل کنید.
* توابع خود را کوتاه و متمرکز بر یک کار واحد تعریف کنید. محدودیت 24 خط سابق همچنان اعمال می شود. اگرچه اندازه و وضوح صفحه نمایش تغییر کرده است، اما از دهه 1960 هیچ چیز درمورد شناخت انسان تغییر نکرده است.
* توابع باید پارامترهای کمی داشته باشند (چهار، یک کران بالایی خوب است). این کار داده‌های ارسال‌شده به توابع را محدود نمی‌کند: گروه‌بندی پارامترهای مرتبط در یک شی واحد، ثابت‌های شی را بومی‌سازی می‌کند و استدلال را با توجه به انسجام و سازگاری آنها ساده می‌کند.
* به طور کلی، هر واحد کد، از یک بلاک تا یک کتابخانه، باید یک اینترفیس محدود داشته باشد. ارتباط کمتر بین اجزا نیاز به استدلال را کاهش می دهد. این بدان معنی است که getter هایی که استیت داخلی را برمی گردانند، دردسرساز هستند - از یک شی برای کار کردن با آن، اطلاعات نخواهید؛ در عوض، از شی بخواهید با اطلاعاتی که از قبل دارد کار را انجام دهد. به عبارت دیگر، کپسوله سازی کاملا در مورد اینترفیس­های محدود است.
* به منظور حفظ متغیرهای کلاس، استفاده از setterها باید ممنوع شود. setterها تمایل دارند به متغیرهایی که بر وضعیت یک شی حاکم هستند اجازه جدا شدن بدهند.
* علاوه بر استدلال در مورد صحت کد، بحث در مورد آن به درک بهتر شما کمک می کند. اطلاعاتی را که به دست می آورید برای همه به اشتراک بگذارید.

**نظری درمورد کامنت­ها**

*کال ایوانز*

در اولین کلاس برنامه نویسی من در کالج، استاد دو برگه کد نویسی BASIC داد. سوال را روی تخته نوشت: «برنامه‌ای بنویسید که بتوانید 10 امتیاز بولینگ را وارد کنید و میانگین امتیازات را بدست آورید.» بعد استاد از کلاس خارج شد. چقدر می تواند سخت باشد؟ راه حل نهایی ام را به خاطر نمی آورم، اما مطمئن هستم که یک حلقه FOR/NEXT در آن وجود داشت و در کل بیشتر از 15 خط کد نبود. ما قبل از اینکه کد را وارد کامپیوتر کنیم، آن را با دست روی برگه می‌نوشتیم – برای هر برگه کدنویسی حدود 70 خط کد مجاز بود. خیلی گیج شده بودم که چرا استاد دو برگه به ما می دهد. از آنجایی که دستخط من همیشه افتضاح بوده، به این امید که چند نمره اضافه برای استایل بگیرم، از دومی برای بازنویسی منظم کدم استفاده کردم.

با کمال تعجب، وقتی در ابتدای کلاس بعد، تکلیف را دریافت کردم، به سختی نمره قبولی را کسب کرده کردم. (این تکلیف قرار بود برای بقیه دوران تحصیلم در کالج نشان­دهنده سواد و دانش من باشد.) در بالای کد کپی شده من نوشته شده بود «هیچ کامنتی نداشتی؟».

این کافی نبود که من و استاد هر دو می دانستیم این برنامه قرار است چه کاری انجام دهد. بخشی از هدف تکلیف این بود که به من بیاموزد کد من باید خودش را به برنامه نویس بعدی که بعد از من می آید، توضیح دهد. این درسی است که هرگز آن را فراموش نکرده­ام.

کامنت­ها نه تنها بد و زیان­آور نیستند بلکه برای برنامه نویسی به اندازه ساختارهای انشعاب پایه یا حلقه ضروری هستند. اکثر زبان‌های مدرن ابزاری شبیه به javadoc دارند که کامنت‌های با فرمت مناسب را برای ساخت خودکار یک سند API تبدیل می‌کنند. این شروع بسیار خوبی است، اما حتی نزدیک به کافی هم نیست. در داخل کد شما باید توضیحاتی وجود داشته باشد که نشان دهد این کد قرار است چه کاری انجام دهد. کدنویسی با ضرب المثل قدیمی، «اگر نوشتن سخت بود، خواندن آن هم باید سخت بود» به مشتری، کارفرما، همکاران و خود آینده شما آسیب می رساند.

از طرف دیگر، ممکن است در کامنت­گذاری زیاده روی کنید. مطمئن شوید که کامنت های شما کد شما را شفاف تر می کنند نه مبهم­تر. کد خود را با کامنت­های مرتبط توسعه دهید و توضیح دهید که کد قرار است چه کاری انجام دهد. کامنت های سرصفحه باید به هر برنامه نویسی اطلاعات کافی را بدهد که بدون نیاز به خواندن کد شما بتواند از کدتان استفاده کند، در حالی که کامنت های بین­خطی شما باید به توسعه دهنده بعدی در اصلاح یا گسترش کد کمک کنند.

یک بار در کاری، با تصمیم کسانی که بالاتر از من بودند، مخالفت کردم. همانند برنامه نویسان جوان، متن ایمیل را که به من دستور می داد از طرح آنها استفاده کنم در بلاک کامنت های هدر فایل چسباندم. معلوم شد که مدیران این فروشگاه در واقع کد نهایی را بررسی کردند. این اولین آشنایی من با اصطلاح تعدیل نیرو بود.

**فقط آنچه را که کد نمی تواند نشان دهد کامنت گذاری کنید**

*کِولین هِنی*

تفاوت بین تئوری و عمل در عمل بیشتر از تئوری است –نکته­ای که مطمئنا برای کامنت­ها هم صدق می­کند. در تئوری، ایده کلی کامنت گذاری کد، ایده باارزشی به نظر می‌رسد: توضیح جزئیات آنچه در حال وقوع است به خواننده. چه کاری می تواند مفیدتر از مفید بودن باشد؟ با این حال، در عمل، کامنت­ها اغلب به یک آسیب تبدیل می شوند. مانند هر شکل دیگری از نوشتن، نوشتن کامنت­ خوب نیز نیاز به مهارت دارد. بیشتر مهارت در این است که بدانید چه زمانی کامنت بنویسید و چه زمانی نیازی به کامنت نیست.

وقتی کد بد شکل و بد ساخت است، کامپایلرها، مفسرها و سایر ابزارها مطمئناً اعتراض خواهند کرد. اگر کد به نحوی از نظر عملکردی نادرست است، بررسی ها، تجزیه و تحلیل استاتیک، تست ها و استفاده روزمره در محیط تولید، اکثر اشکالات را برطرف می کند. اما کامنت ها چطور؟ کِرنیگان و پِلوگر در کتاب *عناصر سبک برنامه نویسی* (Computing McGraw-Hill)، خاطرنشان می‌کنند که «کامنت اگر اشتباه باشد، ارزش صفر (یا منفی) دارند. با این حال، چنین کامنت هایی اغلب در یک کدبیس از زیر دست قسر در می روند، در صورتی که خطاهای کدنویسی هرگز نمی توانند. این کامنت ها منبع ثابتی از حواس پرتی و اطلاعات نادرست هستند، و مانع فکر کردن برنامه­نویس می­شوند.

کامنت هایی که از نظر فنی اشتباه نیستند، اما ارزشی به کد اضافه نمی کنند چه؟ چنین کامنت هایی فقط باعث شلوغی کد می­شوند. کامنت هایی که کد را عیناً تکرار می‌کنند هیچ چیز اضافه­ای به خواننده ارائه نمی‌دهند - بیان چیزی یک بار در کد و یک بار به زبان طبیعی کد را صحیح­تر یا واقعی‌تر نمی‌کند. کدهای کامنت شده کد اجرایی نیستند، بنابراین هیچ اثر مفیدی برای خواننده یا در زمان اجرا ندارند و خیلی سریع منسوخ می شوند. کامنت های مربوط به نسخه و کدهای کامنت شده سعی می کنند به سوالات مربوط به نسخه سازی و تاریخچه پاسخ دهند. این سؤالات قبلاً (به مراتب مؤثرتر) توسط ابزارهای کنترل نسخه پاسخ داده شده است.

رواج کامنت های پر سر و صدا و کامنت های نادرست در یک کدبیس، برنامه نویسان را تشویق می کند تا همه کامنت ها را نادیده بگیرند، چه با رد شدن از آنها یا با پنهان کردنشان. برنامه نویسان زیرک هستند و هر چیزی را که آسیب دیده است دور می زنند: بستن کامنت ها، تغییر رنگ پس زمینه به طوری که کامنت ها و پس زمینه هم رنگ باشند، اسکریپت برای جدا کردن کامنت ها. برای نجات یک کدبیس از چنین کاربردهای نادرستی و کاهش خطر نادیده گرفتن کامنت های با ارزش، باید با کامنت ها به گونه ای برخورد کرد که انگار کد هستند. هر کامنت باید به اطلاعات خواننده بیافزاید، در غیر این صورت اضافی است و باید حذف یا بازنویسی شود.

پس چه کامنتی با ارزش است؟ کامنت ها باید چیزی را که کد نمی گوید نمی تواند بگوید بگویند. کامنتی که توضیح می دهد یک قطعه کد از قبل چه باید بگوید، نشان دهنده این است که ساختار کد یا قراردادهای کدگذاری نیاز به تغییر دارند، بنابراین کد برای خودش صحبت می کند. به جای جبران نام ضعیف متدها یا کلاس ها، نام آنها را تغییر دهید. به جای کامنت گذاری بخش ها در توابع طولانی، توابع کوچکتری را اکسترکت کنید که نام آنها هدف بخش های قبلی را نشان می دهد. سعی کنید تا حد امکان از طریق کد منظورتان را بیان کنید. آنچه را که کد نمی تواند بگوید را کامنت گذاری کنید، نه آنچه را که نمی گوید.

**یادگیری مداوم**

*کلینت شانک*

در دوران جالبی به سر می بریم؛ با پیشرفت فناوری و ارتباطات، رقابت در بازار کار بین‌المللی زیاد شده است و بسیاری از افراد می‌توانند کاری که شما انجام می دهید را انجام دهند. برای حفظ موقعیت خود در بازار کار، لازم است همواره مهارت‌ها و دانش خود را به‌روز کنید. در غیر این صورت به دایناسوری تبدیل می‌شوید که در یک شغل گیر می‌افتید تا روزی که دیگر نیازی به شما نباشد یا شغلتان را نیروی دیگری با درآمد پایین تر تصاحب کند.

خب برای حل این مشکل چه راه حلی دارید؟ برخی از کارفرمایان سخاوتمند هستند و برای پیشرفت شما فرصت های آموزشی فراهم می کنند؛ اما برخی دیگر از کارفرمایان به دلیل کمبود زمان یا منابع مالی قادر نیستند هیچگونه فرصت آموزشی فراهم کنند. پس بهتر است که مسئولیت آموزش خود را خودتان به عهده بگیرید.

در اینجا فهرستی از راه‌هایی برای یادگیری مداوم شما آمده است. بسیاری از این منابع به صورت رایگان در اینترنت قابل دسترسی هستند:

* کتاب ها، مجلات، وبلاگ ها، فیدهای توییتر و وب سایت ها را مطالعه کنید. اگر می خواهید درباره موضوعی اطلاعات بیشتری کسب کنید، به یک لیست پُستی یا گروه خبری بپیوندید.
* اگر واقعاً می خواهید در یک فناوری غرق شوید، باید دست به کار شوید و خودتان کدهای مربوط به آن را بنویسید.
* همیشه سعی کنید با یک منتور کار کنید، زیرا از بقیه بهتر بودن می تواند مانع یادگیری شما شود. اگرچه می‌توانید از هر کس چیزی یاد بگیرید، اما از افراد باهوش‌تر یا با تجربه‌تر از خودتان می‌توانید چیزهای بیشتری یاد بگیرید. اگر نمی توانید منتور پیدا کنید، خودتان به تنهایی یادگیری را ادامه دهید.
* از منتورهای مجازی استفاده کنید. نویسندگان و توسعه دهندگانی را که واقعاً دوست دارید در دنیای مجازی پیدا کنید، وبلاگ های آن ها را دنبال کنید و مطالبی را که می نویسند بخوانید.
* با فریمورک ها و کتابخانه هایی که استفاده می کنید آشنا شوید. وقتی بدانید چیزی چگونه کار می کند می توانید بهتر از آن استفاده کنید. اگر این فریمورک ها و کتابخانه ها اوپن سورس هم باشند، دیگر واقعاً خوش شانس هستید. از دیباگر برای بررسی قدم به قدم کد استفاده کنید تا ببینید در اعماق کد چه خبر است. این کار باعث می شود کدی را که توسط افراد باهوشی نوشته و بازبینی شده است ببینید.
* هر زمان که اشتباهی مرتکب می‌شوید، باگی را برطرف می‌کنید یا با مشکلی مواجه می‌شوید، سعی کنید واقعاً بفهمید که چه اتفاقی افتاده است. به احتمال زیاد شخص دیگری با همین مشکل مواجه شده و آن را در وب پست کرده است. در چنین شرایطی استفاده از گوگل واقعا مفید است.
* یک راه خوب برای یادگیری یک مطلب، تدریس یا صحبت در مورد آن است. وقتی مردم به شما گوش می دهند و از شما سوال می پرسند، انگیزه زیادی برای یادگیری پیدا می کنید. یک بار به صورت آزمایشی ناهارتان را در محل کار، یا یک کنفرانس میل کنید. می‌توانید در محل کار خود یا یک کنفرانس ناهار و یادگیری را امتحان کنید. با برگزاری یک جلسه «ناهار و یادگیری» در محل کار، فرصتی به دست خواهید آورد تا با همکارانتان درباره موضوعی که قصد دارید آموزش دهید، صحبت کنید.
* به یک گروه مطالعهیا یک گروه کاربری محلی برای زبان، فناوری یا رشته ای که به آن علاقه دارید بپیوندید یا خودتان یکی از آن ها راه اندازی کنید.
* در کنفرانس ها حضور داشته باشید ولی اگر امکان حضور در کنفرانس‌ها برای شما وجود ندارد، مشکلی نیست، بسیاری از کنفرانس‌ها ارائه‌های خود را به صورت آنلاین و رایگان در اختیار شما قرار می‌دهند.
* اگر مسیر رفت و آمد شما طولانی است، در راه پادکست گوش دهید.
* آیا تا به حال یک ابزار تجزیه و تحلیل استاتیک را روی پایگاه کد اجرا کرده اید یا به هشدارهای موجود در IDE خود نگاه کرده اید؟ خطاها و هشدارهایی را که این ابزارها نشان می دهند درک کنید.
* توصیه های انجمن برنامه نویسان عملگرا[[1]](#footnote-1) را دنبال کنید و هر سال یک زبان برنامه نویسی و یک تکنولوژی یا ابزار جدید یاد بگیرید. این اقدامات برای شما مسیرهای جدیدی را در دنیای فناوری و برنامه‌نویسی باز می کنند.
* همه چیزهایی که یاد می گیرید نباید در مورد تکنولوژی باشند. افزایش دانش در مورد حوزه کاری خود باعث می‌شود که بهتر بتوانید نیازها و مشکلات کسب و کار را درک کنید و اصلاحات لازم را ارائه دهید. همچنین، یادگیری راهکارها و تکنیک‌های بهبود بهره‌وری و بهتر کار کردن نقش مهمی در پیشرفت شخصی و حرفه‌ای شما دارد.
* به دوران مدرسه و دانشگاه خود برگردید و دروس مربوط به برنامه نویسی را مرور کنید.

خوب است که توانایی‌هایی را که شخصیت نئو در فیلم The Matrix داشت داشته باشیم و به سادگی اطلاعات مورد نیاز خود را در مغزمان دانلود و ذخیره کنیم. اما ما نمی توانیم این کار را کنیم؛ بنابراین برای دستیابی به اطلاعات مورد نیاز، باید زمان و تعهد لازم را صرف یادگیری کنید؛ اما لازم نیست تمام ساعت های بیداری خود را صرف یادگیری کنید. حتی اگر هر هفته کمی زمان برای یادگیری اختصاص دهید، بهتر از این است که هیچ زمانی را به آن اختصاص ندهید. زندگی خارج از محیط کار نیز در جریان است (یا باید جریان داشته باشد).

تکنولوژی به سرعت پیشرفت می کند. از تکنولوژی عقب نمانید.

**راحتی معیار قابل سنجشی نیست**

*گریگور هوپه*

در مورد اهمیت و چالش های طراحی API های خوب، مطالب زیادی گفته شده است. طراحی API برای بار اول دشوار است و اما بعداً تغییر دادن آن بسیار دشوارتر است، درست مثل تربیت فرزندان. اکثر برنامه نویسان باتجربه یاد می گیرند که یک API خوب باید از سطح ثابتی از انتزاع پیروی کند یعنی جزئیات پیاده سازی در API نباید موجود باشد و فقط ویژگی‌ها و عملکردهای مورد نیاز را برای کاربران ارائه دهد، سازگاری و هماهنگی را نشان دهد یعنی نام‌گذاری، اینترفیس ها و رفتارهای API باید منسجم، پیوسته و قابل پیش بینی باشند و اصطلاحات و عباراتی را فراهم کند که به کاربران امکان می‌دهد عملکردهای مختلف را به طرز قابل فهمی بیان کنند و با آنها تعامل داشته باشند. متاسفانه آگاهی از اصول خود به خود منجر به رفتار مناسب نمی شود؛ همانطور که خوردن شیرینی برای سلامتی مضر است، فقط آگاهی داشتن از اصول به تنهایی کافی نیست. برنامه‌نویسان باید این اصول را در عمل به کار ببندند تا یک API خوب و کارآمد طراحی کنند.

به جای موعظه و سخنرانی، می خواهم «استراتژی» طراحی API خاصی را بررسی کنم که بارها و بارها با آن مواجه خواهید شد: استدلال راحتی. این استدلال معمولاً با یکی از «دیدگاه های» زیر شروع می شود:

* نمی‌خواهم کلاس‌های دیگر برای انجام یک کار دو فراخوانی جداگانه انجام دهند.
* چرا باید متد دیگری بسازم اگر تقریباً مشابه این متد است؟ فقط یک switch ساده اضافه می کنم.
* ببینید بسیار ساده است: اگر پارامتر دوم استرینگی با پسوند «.txt» باشد، متد به طور خودکار فرض می کند که پارامتر اول یک filename است، بنابراین واقعاً به دو متد نیاز ندارم.

اگر چه دلایل ارائه شده این آرگومان خوب است، اما چنین آرگومان هایی مستعد کاهش خوانایی کدی هستند که از API استفاده می کند. یک فراخوانی متد مانند:

parser.processNodes(text, false);

بدون اطلاع از پیاده سازی یا حداقل بررسی کردن مستندات عملاً بی معنی است. این متد احتمالاً برای راحتی پیاده کننده (ایمپلمنتر) طراحی شده است، نه فراخوان کننده – عبارت «من نمی‌خواهم فراخوان کننده مجبور باشد دو فراخوانی جداگانه انجام دهد» به «من نمی‌خواستم دو متد جداگانه را کدنویسی کنم» ترجمه شده است. اگر قرار است راحتی پادزهری برای خسته کننده بودن، درهم ریختگی یا ناهماهنگی باشد، اساساً هیچ اشکالی ندارد. با این حال، اگر کمی با دقت در مورد آن فکر کنیم، پادزهر این موارد لزوما راحتی نیست، کارآمدی، ثبات و ظرافت است. APIها قرار است پیچیدگی ها را پنهان کنند، بنابراین می‌توانیم انتظار داشته باشیم که طراحی API خوب نیازمند تلاش است. مطمئناً نوشتن یک متد بزرگ راحت‌تر از مجموعه‌ای از عملیات‌های سنجیده‌شده است، اما آیا استفاده از آن نیز آسان‌تر است؟

با در نظر گرفتن API به عنوان یک زبان می توانیم برای طراحی بهتر API تصمیم بگیریم. یک API با زبانی رسا و قابل فهم، برای پرسیدن سوالات و دریافت پاسخ‌های مفید، واژگان کافی در اختیار کسی که از آن استفاده می کند قرار می دهد. این بدان معنا نیست که یک API برای هر سؤالی که ممکن است ارزش پرسیدن داشته باشد باید دقیقاً یک متد یا یک فعل ارائه کند. واژگان متنوع به ما اجازه می دهد تا مفاهیم پیچیده را به صورت دقیق و کامل در کدهای خود بیان کنیم. مثلا به جای walk(true) ترجیح می‌دهیم بگوییم run، هرچند اساساً می‌توان آن را به عنوان همان عملیات در نظر گرفت که فقط با سرعت‌های مختلف اجرا می‌شود. وقتی واژگان API منظم و با دقت طراحی شود، کد نوشته شده در لایه بالاتر از API قابل فهم تر و رساتر خواهد بود. مهم­تر از آن، واژگان قابل ترکیب به برنامه نویسان دیگر اجازه می دهد تا از API به روش هایی استفاده کنند که ممکن است تا حالا به آن ها فکر هم نکرده باشید - در واقع کار کاربران API را راحت تر می کند. دفعه بعد که وسوسه شدید چند عملیات مشابه را با هم در یک متد API ترکیب کنید، حتی اگر برای عملیات های زیادی کار شما را واقعا راحت تر کند، به یاد داشته باشید که زبان انگلیسی یک کلمه واحد برای MakeUpYourRoomBeQuietAndDoYourHomeWork ندارد.

**استقرار زودهنگام و مکرر**

*استیو برچوک*

دیباگ کردن فرآیندهای استقرار و نصب اغلب تا پایان پروژه به تعویق می افتد. در برخی از پروژه‌ها، نوشتن ابزارهای نصب به مهندس انتشار واگذار می شود و مهندس انتشار نیز این وظیفه را به عنوان یک «کار ضروری سخت» بر عهده می‌گیرد. بررسی ها و نسخه آزمایشی نرم افزار از در یک محیط دستی انجام می شود تا اطمینان حاصل شود که همه چیز درست کار می کند. بنابراین تیم توسعه هیچ تجربه ای در مورد فرآیند یا محیط استقرار یا ندارد و زمانی متوجه مشکلات می شود که ممکن است برای ایجاد تغییرات خیلی دیر شده باشد.

فرآیند نصب و استقرار اولین چیزی است که مشتری می بیند، بنابراین ساده بودن این فرآیندها اولین قدم برای اعتماد کردن مشتریان یا حداقل دیباگ کردن آسان است. نرم افزار مستقر همان چیزی است که مشتری از آن استفاده خواهد کرد. وقتی اطمینان نداشته باشید که استقرار برنامه به درستی انجام شده است یا نه، ممکن است مشتریان شما قبل از استفاده کامل از نرم‌افزار با سوالات و مشکلاتی روبرو شوند.

شروع پروژه با فرآیند نصب به شما زمان می‌دهد تا در طول چرخه توسعه محصول، فرآیند توسعه را تکامل دهید و بتوانید تغییراتی در کد برنامه ایجاد کنید تا نصب آن برای مشتریان آسان‌تر شود. اجرا و تست کردن فرآیند نصب روی محیطی پاک، در فواصل معین، به شما کمک می‌کند تا مشکلات محیطی را شناسایی کنید و اطمینان حاصل کنید که برنامه قابل اجرا و استقرار در محیط‌های مختلف است، بدون اینکه به فرضیات خاصی وابسته باشد که ممکن است در آینده تغییر کنند.

استقرار آخرین مرحله ای است که در فرآیند توسعه مورد توجه قرار می گیرد با این رویکرد، ممکن است کدها بر اساس فرضیاتی طراحی شده باشند که هنگام استقرار منجر به پیچیدگی‌های اضافی شوند. چیزی که در IDE عالی به نظر می رسد، ممکن است فرآیند استقرار بسیار پیچیده تری ایجاد کند چون در IDE کنترل کاملی بر محیط وجود دارد. بنابراین بهتر است در فرآیند استقرار نرم‌افزار تمام موانع و تجارب لازم را هر چه زودتر به دست آوریم.

در ابتدای فرآیند توسعه نرم‌افزار، قابلیت استقرار برنامه بر روی محیط هدف، ممکن است به نظر ارزش تجاری زیادی نداشته باشد؛ اما وقتی بتوانید برنامه را در محیط واقعی اجرا کنید، نیاز دارید کارهای زیادی انجام دهید. در واقع، برای تضمین عملکرد صحیح و قابل قبول برنامه در محیط هدف، نیاز است ابتدا بتوانید آن را استقرار دهید. اگر دلیل شما برای به تعویق انداختن فرآیند استقرار این است که استقرار به نسبت ساده است و در فرآیند توسعه تاثیر بسزایی ندارد، به هر حال پیشنهاد می شود که این کار را انجام دهید زیرا هزینه پایینی دارد. اگر فرآیند استقرار خیلی پیچیده است، یا ابهامات زیادی وجود دارد، همان کاری را که با کد برنامه انجام می دهید انجام دهید: آزمایش کنید، ارزیابی کنید، و حین توسعه، فرآیند استقرار را بهبود بخشید.

فرآیند نصب/استقرار برای بهره‌وری بالاتر مشتریان یا تیم حرفه‌ای شما ضروری است، بنابراین این فرآیند را باید حین توسعه آزمایش کنید و بهبود بخشید. ما در طول پروژه سورس کد را آزمایش و اصلاح می کنیم. فرآیند استقرار نیازمند وقت و توجه است.

**استثناهای کسب و کار را از استثناهای فنی متمایز کنید**

*دَن بِرگ جانسون*

اساساً دو دلیل برای وجود مشکل در زمان اجرا وجود دارد: مشکلات فنی که مانع استفاده از برنامه می­شوند و منطق کسب‌وکار که از سوءاستفاده از برنامه جلوگیری می‌کند. اکثر زبان‌های مدرن، مانند LISP، جاوا، اسمال‌تاک و سی‌شارپ از استثناها برای نشان‌دادن هر دو مشکل استفاده می‌کنند؛ اما این دو استثنا بسیار متفاوت هستند و باید بادقت آن‌ها را از هم تفکیک کرد. نمایش هر دو نوع استثنا با سلسله‌مراتبی یکسان، بدون ذکر کلاس یکسان استثنا، منبع احتمالی سردرگمی است.

اگر خطای برنامه‌نویسی وجود داشته باشد، مشکلات فنی غیرقابل‌حلی رخ می‌دهد. مثلاً تلاش برای دسترسی به عنصر 83 از آرایه‌ای با اندازه 17، خطاست و باید منجر به استثنا شود. نسخه ظریف‌تر این موضوع این است که کد کتابخانه‌ای را با آرگومان‌های نامناسب فراخوانی کنیم که باعث ایجاد همان خطا در داخل کتابخانه می‌شود.

تلاش برای حل این موقعیت‌هایی که خودتان ایجاد کرده‌اید، اشتباه است. در عوض، اجازه می‌دهیم استثنا به بالاترین سطح معماری برسد و سپس مکانیزم کلی مدیریت استثنا، اقدامات لازم برای ایمن‌سازی سیستم را انجام می‌دهد، مانند لغو تراکنش، ثبت خطا و گزارش مناسب به کاربر و مدیر.

یک مورد مشابه این وضعیت زمانی است که در «وضعیت کتابخانه» هستید و یک فراخواننده قرارداد متد شما را نقض کرده است، مثلاً ارسال آرگومان نامعتبر یا تنظیم نادرست شیء وابسته. ارسال آرگومان نادرست مثل دسترسی به عنصر 83 از آرایه 17 عمل می‌کند: فراخواننده باید قبل از فراخوانی، صحت داده‌ها را بررسی می‌کرد. این خطای برنامه‌نویسی سمت کلاینت است و باید با استثنای فنی به آن پاسخ داد.

وضعیتی متفاوت اما همچنان فنی این است که برنامه به دلیل وجود مشکل در محیط اجرا مانند عدم پاسخگویی پایگاه‌داده نمی‌تواند ادامه پیدا کند. در این وضعیت باید فرض کنیم که زیرساخت هر کاری که می‌توانست برای حل مشکل انجام داده است - تعمیر اتصالات و چندین بار تلاش مجدد - ولی ناموفق بوده است. حتی اگر علت متفاوت باشد، وضعیت کد فراخوانی مشابه است: کاری از دستش برنمی‌آید. پس این وضعیت را با استثنا گزارش می‌دهیم تا به مکانیزم کلی مدیریت استثنا برسد.

در مقابل این موارد، وضعیتی وجود دارد که به دلیل منطقی دامنه، نمی‌توانید فراخوانی را تکمیل کنید. این مورد استثنا است؛ یعنی غیرمعمول و نامطلوب است، اما خطای برنامه‌ای نیست و باید با استثنای کسب‌وکار مدیریت شود (مثلاً اگر تلاش کنم پولی از حسابی با موجودی ناکافی برداشت کنم). به‌عبارت‌دیگر، استثناهای کسب‌وکار بخشی از قراردادند که باید استثنا یا سلسله‌مراتب جداگانه‌ای برای آن‌ها تعریف کرد و کلاینت باید منتظر آن‌ها باشد و آمادگی مدیریتشان را داشته باشد.

قراردادن استثناهای فنی و کسب‌وکار در یک سلسله‌مراتب، تمایز بین آن‌ها را مبهم می‌کند. این کار فراخواننده را در مورد قرارداد متد و شرایط فراخوانی و مواردی که باید مدیریت کند، سردرگم می‌کند. جداکردن این موارد ابهامات را رفع می‌کند و احتمال اینکه استثناهای فنی توسط فریمورک برنامه و استثناهای کسب‌وکار توسط کد کلاینت مدیریت شوند، بیشتر می‌شود.

**تمرینات هدفمند زیادی انجام دهید**

*جان جَگِر*

تمرین هدفمند صرفاً انجام یک کار نیست. اگر از خود بپرسید «چرا این کار را انجام می‌دهم؟» و پاسخ شما این است: «برای انجام آن»، در این صورت تمرین هدفمند انجام نمی‌دهید.

تمرین هدفمند برای بهبود مهارت و توانایی انجام کار انجام می‌شود و در مورد مهارت و تکنیک است. تمرین هدفمند شامل تکرار یک کار با هدف افزایش تسلط بر جنبه‌هایی از آن کار است و با تکرارهای مداوم و آهسته تا رسیدن به سطح مطلوب تسلط ادامه می‌یابد. شما تمرین هدفمند را برای تسلط بر کار انجام می‌دهید، نه صرفاً انجام‌دادن آن.

هدف اصلی توسعه پولی و همراه با درآمد، تکمیل یک محصول است، درحالی‌که هدف اصلی تمرین هدفمند بهبود عملکرد شما است. اهداف آن‌ها شبیه هم نیستند. از خود بپرسید چقدر از زمان خود را صرف توسعه محصول دیگران می‌کنید؟ چقدر را صرف توسعه و پیشرفت خودتان؟

چقدر تمرین هدفمند برای کسب تخصص لازم است؟

* پیتر نورویگ نوشته است که «شاید 10000 ساعت... عدد جادویی باشد.»
* مری پاپندیک در کتاب *توسعه نرم‌افزار ناب پیشرو* (انتشارات تخصصی ادیسون-وزلی) خاطرنشان می‌کند که «افراد نخبه نیاز به حداقل 10000 ساعت تمرین هدفمند و متمرکز دارند تا متخصص شوند».

تخصص به تدریج و در طول زمان به دست می‌آید - نه یکباره در ساعت 10000ام! با این وجود، 10000 ساعت بسیار زیاد است: حدود 20 ساعت در هفته به مدت ۱۰ سال. با توجه به این سطح از تعهد، ممکن است به توانایی و استعدادهای خود شک داشته باشید اما نباید داشته باشید. رسیدن به تبحر تا حد زیادی یک انتخاب و تلاش آگاهانه است. تحقیقات در دو دهه اخیر نشان داده است که عامل اصلی در کسب تخصص، زمان صرف­شده برای انجام تمرین هدفمند است؛ توانایی ذاتی عامل اصلی نیست. به گفته مری پاپندیک:

اجماع گسترده‌ای در میان محققان عملکرد متخصص وجود دارد که استعداد ذاتی فقط تا حد آستانه‌ای تأثیر دارد. برای شروع یک ورزش یا حرفه باید حداقل توانایی طبیعی را داشته باشید. پس از آن، افرادی که سخت‌تر کار می‌کنند و تمرین بیش­تری می‌کنند، موفق‌تر خواهند بود.

تمرین هدفمند چیزی که قبلاً در آن متخصص هستید، فایده­ای ندارد. تمرین هدفمند یعنی تمرین چیزی که در آن خوب نیستید و مهارت ندارید. پیتر نورویگ توضیح می دهد:

کلید [پیشرفت در تخصص] تمرین هدفمند است: نه صرفا تکرار. باید خود را با کاری که فراتر از توانایی فعلی شماست به چالش بکشید، آن را امتحان کنید، عملکرد خود را در حین و پس از انجام آن تجزیه و تحلیل کنید و هر گونه اشتباهی را اصلاح کنید.

و مری پاپندیک می‌نویسد:

تمرین هدفمند به معنای انجام کاری که در آن ماهر هستید نیست؛ به معنای به چالش کشیدن خود با انجام کاری که در آن مهارت ندارید است. بنابراین تمرین هدفمند لزوماً سرگرم کننده نیست.

تمرین هدفمند درباره یادگیری است - یادگیری که شما و رفتارتان را تغییر می­دهد. موفق باشید.

**زبان‌های خاص دامنه**

*مایکل هانگر*

هر زمان که به بحث متخصصان در هر حوزه‌ای گوش کنید، اعم از بازیکنان شطرنج، معلمان مهدکودک یا نمایندگان بیمه، متوجه خواهید شد که واژگان آن­ها کاملاً متفاوت از زبان روزمره است. این بخشی از چیزی است که زبان‌های خاص دامنه (DSL) درباره آن صحبت می‌کنند: یک دامنه خاص دارای واژگان تخصصی برای توصیف موارد خاص آن دامنه است.

در دنیای نرم‌افزار، DSLها در مورد عبارات قابل‌اجرا در یک زبان خاص دامنه هستند که از واژگان و دستورزبان محدودی استفاده می‌کنند که برای متخصصان دامنه قابل خواندن، قابل‌فهم و - امیدواریم - قابل نوشتن باشد. DSLهای هدف‌گذاری شده برای توسعه‌دهندگان نرم‌افزار یا دانشمندان از مدت‌ها پیش وجود داشته‌اند. «زبان‌های کوچک» یونیکس که در فایل‌های پیکربندی یافت می‌شوند و زبان‌هایی که با قدرت ماکروهای LISP ایجاد شده‌اند، برخی از نمونه‌های قدیمی‌تر هستند.

DSLها معمولاً به دو دسته *داخلی* و *خارجی* طبقه‌بندی می‌شوند:

DSLهای داخلی

در یک زبان برنامه‌نویسی همه منظوره نوشته شده­اند و سینتکس آن ها به گونه­ای است که بسیار شبیه به زبان طبیعی است. این برای زبان­هایی که ویژگی­هایی برای بهبود خوانایی و فهم کد و قالب بندی بیشتر ارائه می­دهند (مانند روبی و اسکالا) آسان­تر از سایر زبان­هایی است که اینگونه نیستند (مثلاً جاوا). اکثر DSLهای داخلی APIها، کتابخانه‌ها یا کدهای تجاری موجود را رپ می­کنند و رپری (wrapper) برای دسترسی به عملکرد فراهم می­کنند. فقط با اجرای آن­ها مستقیماً قابل اجرا هستند و بسته به پیاده­سازی و دامنه، برای ساخت ساختارهای داده، تعریف وابستگی­ها، اجرای فرآیندها یا وظایف، ارتباط با سایر سیستم­ها یا اعتبارسنجی ورودی کاربر استفاده می­شوند. سینتکس یک DSL داخلی توسط زبان میزبان محدود می­شود. الگوهای زیادی وجود دارد - به عنوان مثال، سازنده عبارت، زنجیره متد، و حاشیه­نویسی - که می­تواند به شما کمک کند تا زبان میزبان را به DSL خود تغییر دهید. اگر زبان میزبان نیازی به کامپایل مجدد نداشته باشد، یک DSL داخلی می­تواند به سرعت در کنار یک متخصص دامنه توسعه پیدا کند.

DSLهای خارجی

عبارت‌های متنی یا گرافیکی زبان هستند - اگرچه DSLهای متنی معمولاً رایج‌تر از DSLهای گرافیکی هستند. عبارات متنی را می­توان توسط یک زنجیره ابزار پردازش کرد که شامل لکسر، تجزیه کننده، ترانسفورماتور مدل، ژنراتور و هر نوع دیگر پس پردازش است. DSLهای خارجی عمدتاً در مدل­های داخلی خوانده می­شوند که اساس پردازش­های بعدی را تشکیل می­دهند. تعریف دستورزبان (به عنوان مثال در EBNF) مفید است. دستورزبان نقطه شروعی را برای تولید بخش­هایی از زنجیره ابزار (به عنوان مثال ویرایشگر، بصری ساز، مولد تجزیه­کننده) فراهم می­کند. برای DSLهای ساده، یک تجزیه کننده دستی ممکن است کافی باشد – مثلا تجزیه کننده ای که از عبارات منظم استفاده می­کند. تجزیه‌کننده‌های سفارشی ممکن است اگر کار زیادی از آن‌ها خواسته شود دست و پاگیر شوند، پس بهتر است ابزارهایی که به طور خاص برای کار با دستور زبان و DSL طراحی شده‌اند را بررسی کرد - مانند openArchitectureWare، ANTLR، SableCC، AndroMDA. تعریف DSLهای خارجی به‌عنوان گویش‌های XML نیز بسیار متداول است، اگرچه خوانایی مخصوصا برای خوانندگان غیرفنی، اغلب مسئله‌ساز است.

شما همیشه باید مخاطبان هدف DSL خود را در نظر بگیرید؛ توسعه دهنده، مدیر، مشتری تجاری یا کاربر نهایی هستند؟ باید سطح فنی زبان، ابزارهای موجود، کمک نحوی (مثلاً IntelliSense)، اعتبارسنجی اولیه، تصویرسازی و نمایش را با مخاطب هدف تطبیق دهید. با پنهان کردن جزئیات فنی، DSL ها می توانند کاربران را با دادن توانایی تطبیق سیستم­ها با نیازهای خود بدون نیاز به کمک توسعه دهندگان توانمند کنند. همچنین می­تواند به دلیل امکان توزیع کار پس از ایجاد فریمورک اولیه زبان، توسعه را تسریع کنند. زبان می­تواند به تدریج تکامل یابد. همچنین مسیرهای مهاجرت متفاوتی برای عبارات و دستورزبان­های موجود در دسترس است.

**از خراب كردن چیزها نترسید**

*مایک لوئیس*

هرکسی که تجربه كار در صنعت نرم­افزار را دارد بدون شک روی پروژه‌ای کار کرده است که کدبیس آن در بهترین حالت بسیار شکننده بوده است. سیستم به طور ضعیفی طراحی شده است و تغییر در یک بخش همیشه موجب خراب‌شدن قسمت دیگری می‌شود که ارتباطی با آن ندارد. هر وقت ماژول جدیدی اضافه می‌شود، هدف برنامه‌نویس این است که تغییرات را به حداقل برساند و در هر انتشار از نتیجه آن بیم و اضطراب داشته باشد. این مثل اين است در یک آسمان‌خراش با تیرهای I شکل بازی جنگا انجام دهیم كه حتماً منجر به فاجعه می‌شود.

دلیل اینکه اعمال تغییرات در چنین سیستمی استرس‌زا و نگران‌کننده است، این است که سیستم بیمار است. این سیستم نیاز به پزشك دارد، در غیر این صورت وضعیت آن فقط بدتر خواهد شد. شما از قبل می‌دانید که مشکل سیستم کجاست، اما از ایجاد تغییرات لازم می‌ترسید. مثل این است که می‌ترسید تخم‌مرغ‌ها را بشکنید تا املت درست کنید. اما یک جراح ماهر می‌داند که برای عمل جراحی باید برش‌هایی ایجاد کرد و این برش‌ها موقتی هستند و التیام خواهند یافت.

از كد خود نترسید. اهمیتی ندارد اگر حین جابه‌جایی اجزاء، موقتاً بخشی از آن خراب شود. ترس فلج‌کننده از تغییر، همان چیزی است که پروژه شما را در ابتدا به این وضعیت دچار کرده است. زمان صرف شده برای بازسازی کدبیس، چندین برابر طول عمر پروژه جبران خواهد شد. یک مزیت اضافی این است که تجربه تیم شما در برخورد با سیستم بیمار، همه شما را متخصص می‌کند تا بدانند سیستم باید چگونه کار کند. به‌جای اینکه از اين دانش متنفر باشید از آن استفاده کنید. کارکردن روی سیستمی که از آن متنفرید، شیوه درستی برای گذراندن زمان هیچ‌کس نیست.

اینترفیس‌های داخلی را مجدداً تعریف کنید، ماژول‌ها را بازسازی کنید، کدهای کپی پیست شده را اصلاح کنید و طراحی خود را با کاهش وابستگی‌ها ساده کنید. می‌توانید پیچیدگی کد را با حذف موارد نادر که اغلب ناشی از اتصال نادرست ویژگی‌ها هستند، به طور قابل‌توجهی کاهش دهید. به‌آرامی ساختار قدیمی را به ساختار جدید تبديل كنيد و حين اين كار تست‌های مختلف انجام دهيد. تلاش برای انجام یک بازسازی بزرگ در یک مرحله، مشکلات زیادی ایجاد می‌کند که ممكن است شما را مجبور می‌کند کل پروژه را در میانه راه رها کنید.

مثل جراحي باشيد كه از بريدن قسمت‌هاي بدن بيمار براي عمل هراس نداشته باشد. این نگرش مسری است و دیگران را ترغیب می‌کند تا روی پروژه‌هایی كه نياز به پاکسازی دارند و به تعویق انداخته‌اند، کار کنند. فهرستی از وظايف «بهداشتي» داشته باشيد که تیم احساس می‌کند برای منافع عمومی پروژه ارزشمند است. مدیریت را متقاعد کنید که اگرچه این وظایف ممکن است نتایج قابل مشاهده‌ای نداشته باشند، اما هزینه‌ها را کاهش و انتشارات آینده را تسهیل می‌کنند. هرگز از توجه به «سلامت» کلی کد غافل نشوید.

**از داده­هاي غيرواقعي يا جالب استفاده نكنيد**

*راد بیگبی*

ديروقت بود. من برای تست طرح‌بندی صفحه‌ای که روی آن کار می‌کردم، داده‌های جایگزین اضافه کرده بودم.

براي نام كاربران از اعضای گروه موسيقي The Clash و براي نام شرکت‌ها از عناوين آهنگ‌های Sex Pistols استفاده كردم. براي نماد سهام فقط چند کلمه چهارحرفی با حروف بزرگ نياز داشتم.

از کلمات ركيك چهارحرفی استفاده کردم.

بی‌ضرر به نظر می‌رسید. فقط براي سرگرمي خودم و شاید دیگر توسعه‌دهندگان روز بعد قبل از اتصال به منبع داده‌های واقعی بود.

صبح روز بعد، یک مدیر پروژه چند اسکرین‌شات برای ارائه گرفت.

تاریخچه برنامه‌نویسی مملو از این نوع داستان‌های جنگی است. کارهایی که توسعه‌دهندگان و طراحان انجام داده‌اند، كارهايي «که هیچ‌کس دیگری نباید می‌دید» اما به طور غیرمنتظره‌ای ديد.

نوع نشت اطلاعات می‌تواند متفاوت باشد، اما زمانی که اتفاق می‌افتد، می‌تواند برای فرد، تیم یا شرکت مسئول مرگبار باشد. مثلاً:

* در طول جلسه، مشتری روی دکمه‌ای کلیک می‌کند که هنوز پیاده‌سازی نشده است. به او گفته می‌شود: «دیگر روی آن کلیک نکن، احمق.»
* به برنامه‌نویسی که از یک سیستم قدیمی نگهداری می‌کند گفته شده است که یک پيغام خطا اضافه کند و تصمیم می‌گیرد از خروجی‌های لاگ‌گیری پشت‌صحنه موجود استفاده کند. ناگهان کاربران با پیام‌هایی مثل «لعنتی، شکست در ذخیره پایگاه‌داده، بتمن!» مواجه می‌شوند.
* کسی ندانسته رابط مدیریتی تست و زنده را قاتى می‌کند و وارد داده‌های «خنده‌دار» می‌کند. مشتریان متوجه «ماساژور شخصی به شکل بیل گیتس به ارزش 1 میلیون دلار» در فروشگاه آنلاین شما می‌شوند.

برای توضیح این موضوع می‌توان از ضرب‌المثل قدیمی استفاده کرد که «دروغ می‌تواند نیمی از جهان را بپیماید درحالی‌که حقیقت کفش‌هایش را به تن کرده است»، در عصر امروز، یک اشتباه می‌تواند قبل از اینکه هر کسی در منطقه زمانی توسعه‌دهنده بیدار شود تا کاری انجام دهد، در توییتر، ردیت و شبکه‌های اجتماعی منتشر شود.

حتی کد منبع شما لزوماً از بررسی‌های دقیق در امان نیست. در سال 2004، وقتی یک فایل فشرده‌شده (tarball) از کد منبع ویندوز 2000 به شبکه‌های اشتراک فایل راه پیدا کرد، برخی افراد با خوشحالی آن را برای یافتن الفاظ رکیک، توهین‌آمیز و دیگر محتوای خنده‌دار grep کردند. (من هم باید بپذیرم که از آن زمان گاهی اوقات کامنت //TERRIBLE HORRIBLE NO GOOD VERY BAD HACK را بکار برده‌ام!)

خلاصه اینکه، هنگام نوشتن هر متنی در کد خود - اعم از کامنت‌ها، لاگ‌ها، ديالوگ ها یا داده‌های تست - همیشه از خود بپرسید اگر این متن عمومی شود چه خواهد شد. این کار باعث صرفه‌جویی در وقت و جلوگیری از شرمندگی‌های بعدی خواهد شد.

**ارورها را ناديده نگيريد!**

*پیت گودلیف*

یک شب که می‌خواستم برای ملاقات دوستانم به یک بار بروم، عجله نداشتم و به مسير نگاه نمی‌کردم. پایم به لبه پیاده‌رو خورد و زمين خوردم. خب، به نظرم چون حواسم نبود حقم بود.

پایم درد گرفت، اما عجله داشتم تا دوستانم را ببينم؛ بنابراین، پس بلند شدم و به راهم ادامه دادم. هرچه بيشتر راه می‌رفتم، دردم بیشتر می‌شد. اگرچه در ابتدا آن را شوک حاصل از سقوط دانسته بودم، به‌سرعت متوجه شدم که مشکلی پیش آمده است.

اما با وجود درد شدید، به راهم ادامه دادم و به بار رفتم. تا زمانی که به آنجا رسیدم، دردم طاقت‌فرسا شده بود. شب خوشی نداشتم، چون حواسم پرت بود. صبح روز بعد نزد دکتر رفتم و متوجه شدم استخوان ساق پایم شکسته است. اگر وقتی درد را احساس کردم ديگر به راهم ادامه نمی‌دادم، از بسیاری آسیب‌های اضافی که با راه‌رفتن روی آن وارد کردم جلوگیری می‌کردم. احتمالاً بدترین صبح پس از شبی سرخوشانه در زندگی من بود.

بسياري از برنامه‌نویسان مثل شب بد من برنامه می‌نویسند.

خطا، چه خطایی؟ جدی نيست، باور کن. می‌توانم آن را نادیده بگیرم. این یک استراتژی برنده برای کدنویسی محکم نیست. در واقع، تنبلی محض است. صرف‌نظر از اینکه فکر می‌کنید خطا در کدتان چقدر بعید است، همیشه و هر بار باید آن را چک و مدیریت کنید. اگر این کار را نکنید، زمان ذخیره نمی‌کنید؛ بلکه مشکلات بالقوه‌ای را برای آینده ذخيره می‌کنید.

ما خطاهای کد خود را به روش‌های مختلفی گزارش می‌کنیم، از جمله:

* **کدهای بازگشتی** می‌توانند به‌عنوان مقدار برگشتی یک تابع برای نشان‌دادن اینکه «کار انجام نشد» استفاده شوند. نادیده‌گرفتن کدهای بازگشت خطا بسیار آسان است. در کد چیزی برای برجسته‌کردن مشکل وجود نخواهد داشت. در واقع، نادیده‌گرفتن مقادیر بازگشتی برخی توابع استاندارد C تبدیل به یک شیوه معمول شده است. چند بار مقدار بازگشتی از printf را چک می‌کنید؟
* **errno** یک متغیر جهانی جداگانه در C است که برای نشان‌دادن خطا تنظیم می‌شود. آسان است که نادیده گرفته شود، استفاده از آن سخت است و منجر به انواع مشکلات ناخوشایند می‌شود. - برای مثال، وقتی چندین thread دارید که تابعي را فراخوانی می‌کنند چه اتفاقی می‌افتد؟ برخی پلتفرم‌ها شما را از این دردسر مصون می‌دارند. اما برخی دیگر نه.
* **استثناها** روش ساختارمندتری هستند که توسط زبان برنامه‌نویسی برای اعلام و مديريت خطاها پشتیبانی می‌شوند. شما نمی‌توانید به‌راحتی آن‌ها را نادیده بگیرید. البته می‌شود! من كدهاي اين چنيني زيادي دیده‌ام:

try {

انجام کاری //

}

catch (...){} نادیده‌گرفتن خطاها//

مزیت این ساختار بد این است که واقعیت انجام کاری اخلاقاً مشکوک را برجسته می‌کند!

اگر خطایی را نادیده بگیرید، چشمانتان را ببندید و وانمود کنید هیچ اشتباهی رخ نداده است، خطرات زیادی را به جان می‌خرید. درست همان‌طور که پای من وضعیت بدتری پیدا کرد؛ چون راه‌رفتن را متوقف نکردم، ادامه‌دادن بدون توجه به علائم خطر می‌تواند منجر به شکست‌های بسیار پیچیده شود. در اولین فرصت به مشكلات رسيدگي كنيد.

عدم رسیدگی به خطاها منجر به اين مشكلات می‌شود:

* **کد شکننده.** كدهايي که پر از باگ‌های هیجان‌انگیز هستند و سخت پيدا می‌شوند.
* **کد ناامن.** کرکرها اغلب از مدیریت ضعیف خطا برای نفوذ به سیستم‌های نرم‌افزاری سوءاستفاده می‌کنند.
* **ساختار ضعیف.** اگر خطاهایی از کد شما وجود دارد که رسیدگی مداوم با آنها خسته‌کننده است، احتمالاً رابط کاربری ضعیفی دارید. آن را طوری طراحي کنید که خطاها کمتر دست‌وپاگیر شوند و رسیدگی به آن­ها آسان‌تر باشد.

همان‌طور که باید تمام خطاهای احتمالی را در کد خود بررسی کنید، باید تمام شرایط بالقوه خطا را در اينترفيس­هاي خود آشکار کنید. آن­ها را پنهان نکنید و طوري وانمود نکنید که سرویس‌های شما همیشه کار می‌کنند. چرا خطاها را بررسی نمی‌کنیم؟ یک سری بهانه‌های رایج وجود دارد:

* رسیدگی به خطاها جریان کد را مختل می‌کند و خوانایی و دنبال‌کردن جریان عادی اجرا را سخت‌تر می‌کند.
* این کار اضافی است و ددلاين من هم نزديك است.
* می‌دانم که این فراخوانی تابع هرگز خطایی را برنخواهد گرداند (printf همیشه کار می‌کند، malloc همیشه حافظه جدید را برمی‌گرداند—اگر اين اتفاق نيفتد، مشکلات بزرگ‌تری داریم...).
* این فقط یک برنامه ابتدایی است و نیازی نیست در سطح تولید نوشته شود.

با کدام یک از اینها موافق هستید؟ چگونه با هر کدام مقابله می‌کنید؟

**زبان را به تنهایی یاد نگیرید، فرهنگ آن را نیز درک کنید**

*آندرس نوراس*

در دبیرستان، مجبور شدم یک زبان عامیانه خارجی یاد بگیرم. در آن زمان، فکر می‌کردم به‌خوبی به زبان انگلیسی مسلط هستم، بنابراین ترجیح دادم در طول سه سالی که کلاس فرانسوی داشتم، سر کلاس بخوابم. چند سال بعد برای تعطیلات به تونس رفتم. عربی زبان رسمی آنجاست و به‌عنوان مستعمره سابق فرانسه، از زبان فرانسوی نیز معمولاً استفاده می‌شود؛ اما انگلیسی فقط در مناطق توریستی صحبت می‌شود. به دلیل ناآگاهی زبانی‌ام، خود را محدود به شناکردن در استخر و خواندن Finnegans Wake، جیمز جویس کردم. این کتاب، نمایشی از توانایی جویس در ادبیات و زبان است. جویس با ترکیب بیش از 40 زبان به شیوه‌ای خلاقانه بازی کرده است. درک اینکه چطور کلمات و عبارات خارجی، راه‌های جدیدی برای بیان خود به نویسنده داده‌اند، تجربه‌ای خسته‌کننده اما غافلگیرکننده بود. من این درس را همراه خودم به حرفه برنامه‌نویسی آورده‌ام.

در کتاب مهم خود با عنوان «برنامه‌نویس عمل‌گرا»، اندی هانت و دیو توماس ما را تشویق می‌کنند که هر سال یک زبان برنامه‌نویسی جدید یاد بگیریم. من سعی کرده‌ام به این توصیه عمل کنم و طی سالیان متعددی تجربه برنامه‌نویسی در زبان‌های مختلفی را داشته‌ام. مهم‌ترین درسی که از این سفر چندزبانه به دست آورده‌ام این است که صرفاً یادگیری دستور زبان برای یادگیری یک زبان کافی نیست، بلکه باید فرهنگ آن زبان را نیز درک کرد.

شما می‌توانید فرترن را در هر زبانی بنویسید، اما برای اینکه واقعاً یک زبان را یاد بگیرید، باید آن را بپذیرید.

اگر کد سی­شارپ شما یک متد Main طولانی با اکثراً متدهای static کمکی است، بهانه نیاورید، بلکه یاد بگیرید چرا کلاس‌ها معنا دارند. از درک سخت عبارت‌های لامبدا که در زبان‌های تابعی استفاده می‌شوند، اجتناب نکنید، بلکه خودتان را مجبور به استفاده از آن‌ها کنید.

هنگامی که قلق یک زبان جدید را یاد گرفتید، شگفت‌زده خواهید شد که چگونه شروع به استفاده از زبان‌هایی که از قبل می‌دانستید، به روش‌های جدید می‌کنید.

من از برنامه‌نویسی با زبان Ruby یاد گرفتم که چگونه در سی‌شارپ از delegates بهتر استفاده کنم. با درک کامل generics در NET. توانستم ایده‌هایی برای بهبود generics در جاوا پیدا کنم و یادگیری LINQ باعث شد یادگیری Scala برایم آسان شود.

همچنین با حرکت بین زبان‌های مختلف، درک بهتری از الگوهای طراحی به دست خواهید آورد. برنامه‌نویسان C متوجه می‌شوند که سی‌شارپ و جاوا الگوی iterator را کالایی کرده‌اند. درحالی‌که در زبان‌های پویا مثل Ruby، ممکن است هنوز از یک visitor استفاده کنید، اما پیاده‌سازی شما شبیه مثال کتاب Gang of Four نخواهد بود.

برخی ممکن است استدلال کنند که Finnegans Wake غیر قابل خواندن است، درحالی‌که برخی دیگر آن را به دلیل زیبایی سبکی­اش تحسین می‌کنند. برای اینکه خواندنش دشوار نباشد، ترجمه‌های تک‌زبانی در دسترس هستند. از قضا، اولین مورد به زبان فرانسوی بود.

کدنویسی هم در بسیاری جهات مشابه است. اگر کد Wakese را با کمی پایتون، مقداری جاوا و کمی هم Erlang بنویسید، پروژه شما به‌هم‌ریخته و شلخته می‌شود. اما اگر به‌جای آن زبان‌های جدید را برای گسترش ذهنتان و گرفتن ایده‌های تازه در مورد راه‌حل‌های متفاوت مسائل امتحان کنید، خواهید دید کدی که در زبان قدیمی محبوبتان می‌نویسید با هر زبان جدیدی که یاد می‌گیرید، زیباتر می‌شود.

**برنامه خود را خیلی سفت‌وسخت و محدود به یک روش خاص ننویسید**

*وِریتی استوب*

من یک‌بار یک آزمون طنز سی پلاس پلاس نوشتم که در آن به طنز، استراتژی زیر را برای مدیریت exceptionها پیشنهاد دادم: با استفاده از بلوک‌های فراوان try...catch در سراسر پایه کدمان، گاهی اوقات قادریم از خاتمه‌یافتن ناگهانی برنامه‌هایمان جلوگیری کنیم. ما حالت نهایی را «میخ‌کردن جسد در وضعیت عمودی» تصور می‌کنیم.

با وجود شوخ‌طبعی من، در واقع داشتم درسی را خلاصه می‌کردم که از تجربه­ای تلخ به دست آورده بودم.

این یک کلاس پایه در کتابخانه سی پلاس پلاس خودمان بود. این کلاس طی سالیان سال توسط انگشتان برنامه‌نویسان مختلف دست‌کاری شده بود: دست هیچ‌کس تمیز نبود. این کلاس حاوی کدی برای کنترل همه exceptionهای فرار کرده از بخش‌های دیگر بود. با الگوبرداری از یوسریان در کتاب Catch-22، تصمیم گرفتیم یا بهتر بگویم احساس کردیم (تصمیم‌گرفتن نشان‌دهنده فکر بیشتری نسبت به آنچه در ساخت این هیولا رفته بود، است) که یک نمونه از این کلاس باید تا ابد زنده بماند یا در این راه بمیرد.

برای این منظور، چندین کنترل‌کننده exception را در هم تنیدیم. ما مدیریت exception ساختاری ویندوز را با مدیریت exception بومی سی پلاس پلاس‌ ترکیب کردیم. وقتی exceptionهای غیرمنتظره‌ای رخ می‌داد، دوباره تلاش می‌کردیم توابع را فراخوانی کنیم و پارامترها را قوی­تر بفرستیم. وقتی به گذشته نگاه می‌کنم، دوست دارم فکر کنم که وقتی در بلوک catch یک کنترل‌کننده دیگر try...catch می‌نوشتم، نوعی آگاهی به من دست می‌داد که به‌اشتباه از بزرگراه عملکرد خوب به کوچه‌ای پر از دیوانگی رفته‌ام. البته این احتمالاً دانش پس از وقوع اتفاق است.

لازم به گفتن نیست که هروقت مشکلی در برنامه‌هایی که بر پایه این کلاس بودند رخ می‌داد، با وجود توابع dump که قرار بود فاجعه را ثبت کنند، مشکلات مثل قربانیان مافیا در کناره دریا ناپدید می‌شدند و هیچ ردی از خود به جا نمی‌گذاشتند تا نشان دهد چه اتفاق لعنتی‌ای افتاده است. در نهایت - پس از مدت زیادی - ما به کاری که انجام داده بودیم نگاه کردیم و شرمنده شدیم. کل آشفتگی را با یک مکانیزم گزارش‌دهی حداقلی و محکم جایگزین کردیم. اما این اتفاق پس از خراب‌کاری‌های زیادی رخ داد.

من نمی‌خواستم شما را با این موضوع خسته کنم - چون قطعاً کس دیگری نمی‌تواند آن‌قدر احمق باشد - اما به‌خاطر بحث آنلاینی که اخیراً با یک نفر داشتم که عنوان شغلی‌اش نشان می‌داد باید بیشتر از اینها می‌دانست، تصمیم گرفتم این موضوع را مطرح کنم. ما در مورد کد جاوا در یک تراکنش از راه دور صحبت می‌کردیم. او استدلال می‌کرد اگر کد شکست بخورد باید exception را در همان‌جا بگیرد و مسدود کند. (من پرسیدم «بعد چه کاری با آن بکند؟ برای شام آن را بپزد؟»)

او قانون طراحان رابط کاربری را نقل کرد: هرگز اجازه ندهید کاربر گزارش خطایی را ببیند، انگار که این کار این موضوع را حل می‌کرد. نمی‌دانم شاید او مسئول کدنویسی خودپردازهایی بوده است که با صفحه آبی مرگ می‌افتادند و به همین دلیل ترومای دائمی گرفته است.

به‌هرحال اگر باید با او ملاقات کردید، درحالی‌که به سمت در می روید، سر تکان دهید و لبخند بزنید و به او توجه نکنید.

**به اینکه «جادو اینجا اتفاق می‌افتد» تکیه نکنید**

*آلن گریفیث*

اگر از دور به هر فعالیت، فرایند یا رشته‌ای نگاه کنید، ساده به نظر می‌رسد. مدیران بی‌تجربه در توسعه فکر می‌کنند کار برنامه‌نویسان ساده است، و برنامه‌نویسان بی‌تجربه در مدیریت هم همین فکر را در مورد کار مدیران دارند.

برنامه‌نویسی چیزی است که بعضی‌ها – بعضی‌اوقات - انجام می‌دهند و قسمت سخت - فکرکردن - کمترین ارزش را برای افراد مبتدی دارد و کمتر هم به چشم می‌آید. تلاش‌های زیادی در طول دهه‌ها برای حذف نیاز به این تفکر ماهرانه صورت گرفته است. یکی از اولین و به یادماندنی‌ترین آنها تلاش گریس هاپر برای غیر مبهم کردن زبان‌های برنامه‌نویسی بود که در برخی گزارش‌ها پیش‌بینی شده بود نیاز به برنامه‌نویس‌های متخصص را از بین می‌برد. نتیجه آن (COBOL) در طول دهه‌های بعدی باعث درآمد زیادی برای بسیاری از برنامه‌نویسان متخصص شده است.

این دیدگاه دائمی که توسعه نرم‌افزار را می‌توان با حذف برنامه‌نویسی ساده کرد، برای برنامه‌نویسی که درگیری‌های آن را درک می‌کند، ساده‌لوحانه است. اما فرایند ذهنی که منجر به این اشتباه می‌شود، بخشی از طبیعت انسان است و برنامه‌نویسان درست مثل همه مردم مستعد انجام آن هستند.

در هر پروژه‌ای، احتمالاً چیزهای زیادی وجود دارد که یک برنامه‌نویس به طور فعال در آن دخالت نمی‌کند: استخراج نیازهای کاربران، تأیید بودجه، راه‌اندازی سرور ساخت، استقرار برنامه به محیط‌های QA و تولید، مهاجرت کسب‌وکار از فرایندها یا برنامه‌های قدیمی و... .

وقتی به طور فعال در کارهایی درگیر نیستید، تمایل ناخودآگاهی وجود دارد که فرض کنید آنها ساده هستند و «با جادو» اتفاق می‌افتند. تا زمانی که جادو ادامه دارد، همه چیز خوب است. اما زمانی که - معمولاً «وقتی» و نه «اگر» - جادو از کار بیفتد، پروژه با مشکل مواجه می‌شود.

من پروژه‌هایی دیده‌ام که به‌خاطر اینکه هیچ‌کس نمی‌دانست چطور وابسته به بارگذاری «نسخه درست» یک DLL هستند، هفته‌ها وقت توسعه‌دهندگان را از دست دادند. وقتی مشکلات متناوباً شروع شد، اعضای تیم تمام جاهای دیگر را بررسی کردند تا اینکه کسی متوجه شد «نسخه اشتباه» DLL بارگذاری می‌شود.

بخش دیگری به‌آرامی در حال اجرا بود - پروژه‌ها به‌موقع تحویل داده می­شدند، نیازی به دیباگ‌کردن شبانه نبود. در واقع، مدیر ارشد به‌تدریج تصمیم گرفت که همه چیز «خوب پیش می‌رود» و می‌تواند بدون مدیر پروژه انجام شود. ظرف شش ماه، پروژه‌های آن بخش دقیقاً مثل بقیه پروژه‌های سازمان شدند - دیر تحویل داده شدند، پر از باگ بودند و به طور مداوم نیاز به patch داشتند.

لازم نیست همه جادویی که باعث می‌شود پروژه شما به نتیجه برسد را درک کنید، اما درک بخشی از آن یا قدردانی از کسی که چیزهایی را که شما نمی‌دانید درک می‌کند، ضرری ندارد. مهم‌تر از همه، مطمئن شوید که وقتی جادو از کار می‌افتد، می‌توان آن را دوباره راه انداخت.

**خودت را تکرار نکن**

*استیو اسمیت*

از میان تمام اصول برنامه‌نویسی، اصل «خودت را تکرار نکن[[2]](#footnote-2)» (DRY) احتمالاً یکی از بنیادی‌ترین آن‌هاست. این اصل توسط اندی هانت و دیو توماس در کتاب The Pragmatic Programmer مطرح شده است و در زیربنای بسیاری دیگر از بهترین شیوه‌ها و الگوهای معروف توسعه نرم‌افزار قرار دارد. برنامه‌نویسی که یاد بگیرد تکرار را شناسایی کند و با استفاده از شیوه‌ها و انتزاع مناسب آن را حذف نماید، می‌تواند کدی تمیزتر از کسی تولید کند که به طور مداوم برنامه را با تکرارهای غیرضروری آلوده می‌کند.

**تکرار، هدردادن منابع است**

هر خط کدی که به برنامه اضافه می‌شود، نیاز به نگهداری دارد و منبع بالقوه‌ای برای بروز باگ‌های آینده است. تکرار بی‌دلیل، کدبیس را بیش از حد بزرگ می‌کند و منجر به بروز باگ­های بیشتر و اضافه‌کردن پیچیدگی اتفاقی به سیستم می‌شود. حجم اضافه شده به سیستم باعث می‌شود توسعه‌دهندگانی که با سیستم کار می‌کنند، درک کاملی از کل سیستم نداشته باشند یا مطمئن نباشند که تغییرات انجام شده در یک مکان نیاز به اعمال در مکان‌های دیگری ندارد که منطق مشابهی دارند. اصل DRY می‌گوید «هر دانشی باید نماینده‌ای یکتا، قاطع و معتبر در سیستم داشته باشد.»

**تکرار در فرایندها نیازمند خودکارسازی است**

اصل DRY در این موارد نیز مانند سورس­کد برنامه کاربرد دارد. تست­های دستی کند، خطاپذیر و تکرارناپذیر هستند، بنابراین جایی که امکان دارد باید از مجموعه­های تست خودکار استفاده کرد. ادغام نرم‌افزار به‌صورت دستی زمان‌بر و خطاپذیر است، پس فرایند ساخت تاحدامکان به طور مکرر و ایده‌آل باید با هر چک‌این اجرا شود. هرجا فرایندهای دستی دردسرسازی هست که قابلیت خودکارسازی دارند، باید خودکارسازی و استانداردسازی شوند. هدف، اطمینان از وجود تنها یک راه برای انجام کار، و آن هم آسان‌ترین راه ممکن است.

**تکرار در منطق نیازمند انتزاع است**

تکرار در منطق می‌تواند اشکال مختلفی داشته باشد. کپی و پیست منطق if-then یا switch-case از جمله راحت‌ترین‌ها روش­ها برای تشخیص و اصلاح است. بسیاری از الگوهای طراحی با هدف صریح کاهش یا حذف تکرار در منطق درون برنامه ایجاد شده‌اند. اگر معمولاً یک شی قبل از استفاده نیاز به انجام چند کار دارد، این کار با الگوهای Abstract Factory یا Factory Method انجام می‌شود. اگر یک شیء رفتارهای متفاوتی دارد، این رفتارها با الگو Strategy تزریق می‌شوند تا از ساختارهای بزرگ if-then جلوگیری شود. در واقع، فرموله‌کردن خود الگوهای طراحی تلاشی برای کاهش تکرار تلاش موردنیاز برای حل مشکلات رایج و بحث در مورد چنین راه‌حل است. علاوه بر این، DRY می‌تواند به ساختارهایی مانند طرحواره پایگاه‌داده‌ اعمال شود که این کار منجر به نرمال‌سازی می‌شود.

**مهم اصول است**

سایر اصول نرم‌افزاری نیز با DRY مرتبط هستند. اصل Once and Only Once «فقط یکبار» که فقط به رفتار عملکردی کد اعمال می‌شود، می‌تواند بخشی از DRY در نظر گرفته شود. اصل باز/بسته[[3]](#footnote-3) که می‌گوید «موجودیت‌های نرم‌افزاری (کلاس‌ها، ماژول‌ها و عملکردها) باید برای توسعه باز و برای تغییر بسته باشند» تنها زمانی در عمل کار می‌کند که DRY رعایت شده باشد. همچنین، اصل معروف مسئولیت واحد[[4]](#footnote-4) که نیازمند این است که یک کلاس «فقط یک دلیل برای تغییر داشته باشد»، به DRY وابسته است.

وقتی در مورد ساختار، منطق، فرایند و عملکرد رعایت شود، اصل DRY راهنمایی بنیادی به توسعه‌دهندگان نرم‌افزار ارائه می‌دهد و به ایجاد برنامه‌های ساده‌تر، قابل نگهداری و با کیفیت بالاتر کمک می‌کند. اگرچه سناریوهایی وجود دارد که تکرار برای برآورده‌کردن نیازمندی‌های عملکردی یا سایر نیازمندی‌ها (به‌عنوان‌مثال عدم نرمال‌سازی داده در پایگاه‌داده) ضروری است، اما تنها در جایی باید استفاده شود که به طور مستقیم مشکل واقعی را حل کند، نه چیزی را که فقط مشکل پنداشته می‌شود.

**به کد دست نزنید!**

*کال ایوانز*

این اتفاق برای خیلی از توسعه‌دهندگان پیش آمده است. کد شما برای تست سیستم به سرور مرحله‌بندی ارسال شده و مدیر تست به شما گزارش می‌دهد که مشکلی پیدا کرده است. اولین واکنش شما این است: «اجازه دهید سریع مشکل را رفع کنم، می‌دانم مشکل از کجاست».

اما در نگاه کلی‌تر، اشتباه اینجاست که به‌عنوان یک توسعه‌دهنده، فکر می‌کنید باید به سرور مرحله‌بندی دسترسی مستقیم داشته باشید.

در اکثر محیط‌های توسعه مبتنی بر وب، معماری را می‌توان به‌صورت زیر تقسیم­بندی کرد:

• توسعه محلی و تست واحد روی ماشین توسعه‌دهنده

• سرور توسعه که در آن تست یکپارچه‌سازی دستی یا خودکار انجام می‌شود

• سرور مرحله‌بندی که در آن تیم QA و کاربران، تست پذیرش را انجام می‌دهند

• سرور تولید

بله سرورها و سرویس‌های دیگری مانند کنترل سورس­کد و ثبت موارد نیز وجود دارند، اما ایده کلی را متوجه می‌شوید. با استفاده از این مدل، یک توسعه‌دهنده - حتی یک توسعه‌دهنده ارشد - هرگز نباید به فراتر از سرور توسعه دسترسی داشته باشد. اکثر توسعه‌ها بر روی ماشین محلی توسعه‌دهنده با استفاده از ترکیب موردعلاقه‌ای از IDE ها، ماشین‌های مجازی و برای شانس خوب مقدار مناسبی جادوی سیاه انجام می‌شود.

پس از بررسی کد در سیستم کنترل نسخه (SCC)، چه به‌صورت خودکار یا دستی، کد باید به سرور توسعه منتقل شود. در آنجا می‌توان تست‌های لازم را انجام داد و در صورت نیاز تغییراتی اعمال کرد تا اطمینان حاصل شود همه چیز در کنار هم درست کار می‌کند. از این مرحله به بعد، توسعه‌دهنده تنها ناظر فرایند است.

مدیر مرحله‌بندی باید کد را ارائه و به سرور مرحله‌بندی برای تیم QA ارسال نماید. همان‌طور که توسعه‌دهندگان نیازی به دسترسی فراتر از سرور توسعه ندارند، تیم QA و کاربران نهایی هم نیازی به دستکاری چیزی در سرور توسعه ندارند. اگر کد برای تست پذیرش آماده است، باید نسخه­ای جدید منتشر شود. نباید از کاربر خواست به‌صورت موقتی چیزی را در سرور توسعه بررسی کند، مگر اینکه تنها توسعه‌دهنده پروژه باشید؛ زیرا افراد دیگری هم کدهایشان در آنجا است و ممکن است آمادگی نشان‌دادن به کاربر را نداشته باشند. تنها مدیر انتشار باید به هر دو سرور دسترسی داشته باشد.

تحت هیچ شرایطی - هرگز، اصلاً - یک توسعه‌دهنده نباید به سرور تولید دسترسی داشته باشد. اگر مشکلی وجود دارد، کارکنان پشتیبانی باید آن را رفع کنند یا درخواست کنند شما آن را رفع کنید. پس از بررسی آن در SCC، آنها یک patch از آنجا ارسال خواهند کرد. برخی از بزرگ‌ترین فاجعه‌های برنامه‌نویسی که من درگیر آن بوده‌ام، زمانی رخ داده است که کسی (در اصل خودم) این قانون آخر را نقض کرده است. اگر خراب است، تولید جایی برای تعمیر آن نیست.

**رفتار و منطق را کپسوله کنید، نه فقط استیت را**

*اینار لاندر*

در تئوری سیستم‌ها، محصورسازی یکی از مفیدترین ساختارها برای کار با سیستم‌های بزرگ و پیچیده است. در صنعت نرم‌افزار، ارزش محصورسازی یا کپسوله‌سازی به‌خوبی درک شده است. محصورسازی توسط ساختارهای زبان برنامه‌نویسی مانند زیرمجموعه­ها و توابع، ماژول‌ها و پکیج‌ها، کلاس‌ها و غیره پشتیبانی می‌شود.

ماژول‌ها و پکیج‌ها نیازهای کپسوله‌سازی در مقیاس بزرگ‌تر را برآورده می‌کنند، درحالی‌که کلاس‌ها، زیرمجموعه­ها و توابع به جنبه‌های ظریف‌تر این موضوع می‌پردازند. طی سال‌ها متوجه شده‌ام کلاس‌ها به نظر یکی از سخت‌ترین ساختارهای کپسوله‌سازی برای توسعه‌دهندگان برای پیاده‌سازی درست آن هستند. یافتن کلاسی با یک متد اصلی 3000 خطی یا کلاسی فقط با متدهای set و get برای ویژگی‌های اولیه آن غیرمعمول نیست. این مثال‌ها نشان می‌دهند که توسعه‌دهندگان کاملاً تفکر شیءگرا را درک نکرده‌اند و از قدرت اشیا به‌عنوان ساختارهای مدل‌سازی استفاده نکرده‌اند. برای توسعه‌دهندگان آشنا با اصطلاحات POJO[[5]](#footnote-5) و POCO[[6]](#footnote-6)، هدف، بازگشت به اصول اولیه شیءگرایی به‌عنوان یک پارادایم مدل‌سازی همین بود - اشیا ساده و معمولی هستند، نه گنگ.

یک شیء هم استیت و هم رفتار را کپسوله می‌کند که در آن رفتار بر اساس استیت واقعی تعریف می‌شود. یک درب را در نظر بگیرید. درب چهار حالت دارد: بسته، باز، در حال بسته‌شدن، در حال باز شدن. این حالت ها دو عملیات را ارائه می‌دهند: باز کردن و بستن. بسته به استیت، عملیات باز و بسته متفاوت رفتار می‌کنند. این ویژگی ذاتی یک شیء، فرایند طراحی را مفهوماً ساده می‌کند. آن را در دو وظیفه ساده خلاصه می‌کند: تخصیص و تفویض مسئولیت به اشیای مختلف از جمله پروتکل‌های تعامل بین شیء.

این که این موضوع چگونه در عمل کار می‌کند، بهتر است با یک مثال توضیح داده شود. فرض کنید سه کلاس داریم: Customer، Order و Item. شیء Customer مکان مناسبی برای نگهداری محدوده اعتبار و قوانین اعتبارسنجی است. شیء Order از Customer مربوط به خودش اطلاع دارد و با فراخوانی متد customer.validateCredit(item.price()) بررسی واقعی اعتبار را به آن واگذار می‌کند. اگر شرط بعدی برای متد رعایت نشود، می‌توان یک استثنا ایجاد کرد و خرید را لغو کرد.

توسعه‌دهندگان کم‌تجربه‌تر ممکن است تصمیم بگیرند کلیه منطق کسب‌وکار را در یک شیء رپ کنند که اغلب OrderManager یا OrderService نامیده می‌شود. در این طراحی ها باOrder ، Customer و Item مثل انواع رکورد رفتار می شود. کل منطق از کلاس‌ها خارج و در یک متد بزرگ رویه‌ای با ساختارهای درونی if-then-else زیاد به هم متصل می‌شود. این متدها به‌راحتی خراب می‌شوند و تقریباً غیر قابل نگهداری هستند. دلیل چیست؟ کپسوله‌سازی نقض شده است.

بنابراین، در نهایت کپسوله‌سازی را رعایت کنید و از قابلیت‌های زبان برنامه‌نویسی خود برای حفظ آن استفاده کنید.

**اعداد اعشاری واقعی نیستند**

*چاک آلیسون*

اعداد اعشاری در مفهوم ریاضی «اعداد واقعی» نیستند، اگرچه در برخی زبان‌های برنامه‌نویسی مانند پاسکال و فرترن واقعی نامیده می‌شوند. اعداد حقیقی دقت نامحدود دارند؛ بنابراین پیوسته و بدون ازدست‌دادن اطلاعات هستند. درحالی‌که اعداد اعشاری دقت محدودی دارند؛ بنابراین متناهی هستند و شبیه اعداد صحیح «بد و نامنظم» رفتار می‌کنند، زیرا به طور یکنواخت در محدوده خود توزیع نشده‌اند.

به‌عنوان‌مثال اگر 2147483647 (بزرگ‌ترین عدد صحیح 32 بیتی) را به یک متغیر اعشاری 32 بیتی (مثلاً x) اختصاص دهیم و چاپ کنیم، 2147483648 را مشاهده می‌کنیم. حالا اگر x-64 را چاپ کنیم باز 2147483648 به دست می‌آید. وقتی x-65 را چاپ کنیم، 2147483520 را خواهیم دید! چرا؟ زیرا فاصله بین اعداد اعشاری مجاور در آن محدوده، 128 است و عملیات اعشاری به نزدیک‌ترین عدد اعشاری گرد می‌شوند.

اعداد اعشاری در استاندارد IEEE بر پایه نماد علمی مبنای دو، دارای دقت ثابت هستند: 1.d1d2...dp × 2^e که در آن p دقت (24 برایfloat و 53 برای double) است. فاصله بین دو عدد اعشاری متوالی 2^(-p+e) است که می‌توان آن را با ε|x| تقریب زد که در آن ε خطای ماشین (2^(-p)) است.

دانستن فاصله در همسایگی یک عدد اعشاری به جلوگیری از خطاهای رایج عددی کمک می‌کند. مثلاً در محاسبات تکراری مانند جستجوی ریشه معادله، درخواست دقت بیش از آنچه سیستم اعداد می‌تواند در همسایگی پاسخ بدهد، بیهوده است. مطمئن شوید تلرانس درخواستی کوچک‌تر از فاصله آنجا نباشد وگرنه تا ابد حلقه خواهید زد.

ازآنجایی‌که اعداد اعشاری تقریب‌هایی از اعداد واقعی هستند، به طور اجتناب‌ناپذیری کمی خطا وجود دارد. این خطا که گردشدگی نام دارد می‌تواند منجر به نتایج غیرمنتظره‌ای شود.

وقتی اعداد تقریباً برابر را از هم کم می‌کنید، مهم‌ترین ارقام یکدیگر را حذف می‌کنند، بنابراین کم‌اهمیت‌ترین رقم (که خطای گردشدگی در آن وجود دارد) به موقعیت بیشترین اهمیت در نتیجه اعشاری ارتقا می‌یابد و در نتیجه محاسبات بعدی مرتبط را آلوده می‌کند (پدیده‌ای به نام پخش‌شدگی). شما باید الگوریتم‌های خود را برای جلوگیری از چنین لغو شدگی فاجعه‌باری بررسی کنید. به‌عنوان‌مثال در حل معادله درجه دوم x^2 - 100000x + 1 = 0 با فرمول ریشه‌یابی، چون عملوندهای -b و sqrt(b^2 - 4) تقریباً برابرند، به‌جای محاسبه‌ی -b + sqrt(b^2 - 4) می‌توان ریشه‌ی r1 = -b - sqrt(b^2 - 4) را محاسبه کرد و باتوجه‌به رابطه‌ی r1\*r2 = c/a در معادله‌ی درجه دوم، r2 را به دست آورد.

پخش‌شدگی می‌تواند به شکل‌های پیچیده‌تری هم رخ دهد. فرض کنید یک کتابخانه به طور ساده‌لوحانه ex را با فرمول 1 + x + x^2/2 + x^3/3! + ... محاسبه می‌کند. این برای x مثبت خوب کار می‌کند، اما در نظر بگیرید چه اتفاقی می‌افتد وقتی x یک عدد منفی بزرگ است. جملات توان زوج منجر به اعداد مثبت بزرگی می‌شوند و کم‌کردن مقادیر توان‌های فرد اصلاً تأثیری بر نتیجه ندارد. مشکل اینجاست که گردشدگی در جملات مثبت بزرگ، در مرتبه ارزش مکانی بسیار بالاتری نسبت به جواب واقعی قرار دارد. جواب به سمت بی‌نهایت مثبت میل می‌کند! راه‌حل هم ساده است: برای x منفی، ex = 1/e^|x| را محاسبه کنید.

البته نباید گفت که نباید از اعداد اعشاری برای کاربردهای مالی استفاده کرد - برای این منظور از کلاس‌های اعشاری در زبان‌هایی مانند پایتون و C# استفاده می‌شود. اعداد اعشاری برای محاسبات علمی کارآمد در نظر گرفته شده‌اند. اما کارایی بدون دقت بی‌فایده است، بنابراین منبع خطاهای گردشدگی را به یاد داشته باشید و متناسب با آن کدنویسی کنید!

**آرزوهایتان را با پروژه­های اوپن سورس برآورده کنید**

*ریچارد مونسون-هفل*

احتمالاً شما در محل کارتان در حال توسعه نرم‌افزارهایی هستید که با آرزوها و رؤیاهای بلندپروازانه‌تان در زمینه توسعه نرم‌افزار مطابقت ندارند. ممکن است در حال توسعه نرم‌افزاری برای یک شرکت بیمه بزرگ باشید، درحالی‌که ترجیح می‌دهید در گوگل، اپل، مایکروسافت یا روی استارتاپ خودتان کار کنید و روی پروژه‌های بزرگ و نوآورانه‌ای متمرکز باشید. اگر روی سیستم‌ها و نرم‌افزارهایی کار کنید که به آن‌ها علاقه‌ای ندارید، هرگز به جایی که می‌خواهید نخواهید رسید.

خوشبختانه برای حل مشکل شما یک راه‌حل وجود دارد: اوپن سورس. هزاران پروژه اوپن سورس فعال وجود دارد که می‌توانند هر نوع تجربه توسعه نرم‌افزاری که مدنظر شماست را فراهم کنند. اگر به توسعه سیستم‌عامل علاقه‌مند هستید، می‌توانید در یکی از ده‌ها پروژه اوپن سورس سیستم‌عامل مشارکت کنید. اگر می‌خواهید روی نرم‌افزارهای موسیقی، انیمیشن، رمزنگاری، رباتیک، بازی‌های رایانه‌ای، بازی‌های آنلاین چندنفره، موبایل و غیره کار کنید، قطعاً چندین پروژه اوپن سورس پیدا خواهید کرد که به این حوزه‌ها اختصاص دارند.

البته هیچ‌چیزی بدون تلاش و رایگان به دست نمی‌آید. شما باید حاضر باشید وقت آزاد خود را اختصاص دهید، چرا که احتمالاً نمی‌توانید در محل کارتان روی یک بازی ویدیویی اوپن سورس کار کنید - هنوز در قبال کارفرمایتان مسئولیتی دارید. علاوه بر این، تعداد کمی از افراد از مشارکت در پروژه‌های اوپن سورس درآمدزایی می‌کنند - البته برخی افراد درآمدزایی می‌کنند؛ اما اکثریت افراد نه. باید حاضر باشید مقداری از وقت آزادتان را اختصاص دهید (کمتر بازی‌کردن و تماشای تلویزیون آسیبی به شما نمی‌زند). هرچه بیشتر روی یک پروژه اوپن سورس کار کنید، سریع‌تر به آرزوهای واقعی‌تان به‌عنوان یک برنامه‌نویس دست خواهید یافت. همچنین درنظرگرفتن و توجه به قرارداد استخدامی‌تان بسیار مهم است - برخی کارفرمایان ممکن است مشارکت شما را محدود کنند، حتی در زمان­های آزادتان. علاوه بر این باید از نقض قوانین مالکیت معنوی مرتبط با حق تکثیر، ثبت اختراع، علائم تجاری و اسرار تجاری اجتناب کنید.

اوپن سورس فرصت‌های عظیمی را برای برنامه‌نویسان باانگیزه فراهم می‌کند. اول اینکه می‌توانید ببینید دیگران چگونه یکی از راه‌حل­های موردعلاقه شما را پیاده‌سازی کرده‌اند - می‌توانید با خواندن سورس­کد دیگران چیزهای زیادی یاد بگیرید. دوم اینکه می‌توانید کد و ایده‌های خودتان را به پروژه اضافه کنید - البته هر ایده خوبی که داشته باشید پذیرفته نمی‌شود، اما برخی پذیرفته خواهند شد و شما حتی با کار روی راه‌حل‌ها و ارائه کد، چیزهای جدیدی یاد خواهید گرفت. سوم اینکه با افراد عالی آشنا می‌شوید که علاقه مشترکی به نوع مشخصی از نرم‌افزار دارند - این دوستی‌های اوپن سورس می‌توانند مادام‌العمر باشند و در نهایت اگر مشارکت‌کننده شایسته‌ای باشید، می‌توانید تجربه واقعی دنیای واقعی را در حوزه فناوری موردعلاقه‌تان کسب کنید.

شروع کار با پروژه‌های اوپن سورس در واقع کار ساده‌ای است. منابع زیادی در مورد ابزارهای موردنیاز (مدیریت سورس­کد، ویرایشگرها، زبان‌های برنامه‌نویسی، سیستم‌های بیلد و غیره) وجود دارد. ابتدا پروژه‌ای را که می‌خواهید روی آن کار کنید، پیدا کنید و با ابزارهایی که آن پروژه استفاده می‌کند، آشنا شوید. مستندات پروژه‌ها معمولاً کم‌حجم هستند، اما این موضوع چندان مهم نیست؛ چون بهترین راه یادگیری بررسی مستقیم سورس­کد است. اگر می‌خواهید مشارکت داشته باشید، می‌توانید پیشنهاد دهید در مستندسازی کمک کنید. یا می‌توانید با نوشتن کد تست شروع کنید. اگرچه این کار هیجان‌انگیز به نظر نمی‌رسد، واقعیت این است که با نوشتن کد تست برای نرم‌افزار دیگران سریع‌تر یاد می‌گیرید تا هر فعالیت دیگری در توسعه نرم‌افزار. پس کد تست خوب، واقعاً خوب، بنویسید. باگ‌ها را پیدا کنید، راه‌حل‌هایی پیشنهاد دهید، دوستانی پیدا کنید، روی نرم‌افزار موردعلاقه‌تان کار کنید و به آرزوهایتان در توسعه نرم‌افزار جامه عمل بپوشانید

**قانون طلایی طراحی API**

*مایکل فیرز*

طراحی API به‌ویژه برای مقیاس بزرگ کار دشواری است. اگر API ای طراحی می‌کنید که قرار است صدها یا هزاران کاربر داشته باشد، باید در مورد این فکر کنید که چگونه ممکن است در آینده آن را تغییر دهید و این تغییرات ممکن است کلاینت کد را خراب کند. علاوه بر این، باید در مورد اینکه چگونه کاربران API شما روی شما تأثیر می‌گذارند نیز فکر کنید. اگر یکی از کلاس‌های API شما به‌صورت داخلی از یکی از متدهای خودش استفاده کند، باید به یاد داشته باشید که یک کاربر می‌تواند از کلاس شما مشتق شود و آن متد را override کند که این می‌تواند فاجعه‌آمیز باشد. دیگر نمی‌توانید آن متد را تغییر دهید؛ چون برخی از کاربران معنای دیگری به آن داده‌اند. انتخاب‌های پیاده‌سازی داخلی آینده شما در اختیار کاربران است.

توسعه‌دهندگان API به روش‌های مختلفی این مشکل را حل می‌کنند، اما راحت‌ترین راه قفل‌کردن API است. اگر با جاوا کار می‌کنید، شاید تمایل داشته باشید اکثر کلاس‌ها و متدهایتان را final کنید. در C#‌، شاید کلاس‌ها و متدها را sealed کنید. صرف‌نظر از زبانی که استفاده می‌کنید، شاید تمایل داشته باشید API خود را از طریق یک سینگلتون یا متدهای استاتیک فابریک ارائه دهید تا از آن را در برابر افرادی محافظت کنید که ممکن است رفتار را override کنند و از کد شما به روش‌هایی استفاده کنند که ممکن است انتخاب‌های آینده شما را محدود کند. همه اینها منطقی به نظر می‌رسد، اما آیا واقعاً چنین است؟

در طول دهه گذشته، به‌تدریج دریافته‌ایم که تست واحد بخش بسیار مهمی از تمرینات است، اما این درس هنوز به طور کامل در صنعت نفوذ نکرده است. شواهد آن در اطراف ما وجود دارد. یک کلاس تست نشده دلخواه که از API شخص ثالث استفاده می‌کند را در نظر بگیرید و سعی کنید برای آن تست واحد بنویسید. اکثر مواقع با مشکل مواجه خواهید شد. متوجه می‌شوید کدی که از API استفاده می‌کند، مثل چسب به آن متصل است. هیچ راهی برای جعل کلاس‌های API وجود ندارد تا بتوانید تعاملات کد خود را با آن‌ها حس کنید یا مقادیر بازگشتی را برای تست تأمین کنید.

با گذشت زمان، این وضعیت بهتر خواهد شد، اما تنها درصورتی‌که هنگام طراحی API، تست را به‌عنوان یک استفاده واقعی در نظر بگیریم. متأسفانه این کار فراتر از صرفاً تست‌کردن کد خودمان است. جایی که قانون طلایی طراحی API اهمیت پیدا می‌کند این است که صرفاً نوشتن تست برای یک API توسعه داده شده کافی نیست؛ باید تست‌های واحدی نیز برای کدی که از API شما استفاده می‌کند، بنویسید. زمانی که از این قانون پیروی کنید، از نزدیک موانعی را تجربه خواهید کرد که کاربرانتان باید برای تست مستقل کدشان برطرف کنند.

برای توسعه‌دهندگان راه مشخصی برای آسان‌کردن تست کدی که از API شما استفاده می‌کند، وجود ندارد. static، final و sealed ذاتاً ساختارهای بدی نیستند، گاهی می‌توانند مفید باشند؛ اما مهم است که از موضوع تست‌پذیری آگاه باشید و برای این کار باید خودتان آن را تجربه کنید. وقتی آن را تجربه کردید، می‌توانید مانند هر چالش طراحی دیگری به آن بپردازید.

**افسانه گورو**

*رایان براش*

هر کس که مدت زمان کافی در حوزه نرم‌افزار کار کرده باشد، حتماً سؤالاتی نظیر این را شنیده است:

من استثنای XYZ را دریافت می‌کنم. می‌دانی مشکل از کجاست؟

کسانی که این سؤال را می‌پرسند به‌ندرت به خود زحمت می‌دهند ردیابی استک، لاگ‌های خطا یا هر زمینه‌ای که منجر به مشکل می‌شود را در نظر بگیرند. انگار فکر می‌کنند شما در سطحی متفاوت کار می‌کنید و راه‌حل‌ها بدون تجزیه‌وتحلیل مبتنی بر شواهد به سراغتان می‌آیند. آنها فکر می‌کنند شما یک گورو هستید.

انتظار چنین سؤالاتی از افرادی که با نرم‌افزار آشنایی ندارند، طبیعی است. برای آنها سیستم‌ها تقریباً جادویی به نظر می‌رسند. چیزی که نگران‌کننده این است که چنین سؤالاتی را در خود جامعه نرم‌افزاری هم می‌بینیم. مثلاً سؤالات مشابهی در طراحی برنامه مطرح می‌شود: «من در حال ساخت سیستم مدیریت انبار هستم. آیا باید از قفل خوش‌بینانه استفاده کنم؟» طنز ماجرا این است که اغلب کسانی که سوال می‌پرسند، خودشان از پاسخ‌دهنده بیشتر توانایی پاسخگویی دارند. سؤال‌کنندگان احتمالاً کلیت موضوع را می‌دانند، نیازمندی‌ها را می‌شناسند و می‌توانند درباره مزایا و معایب راهبردهای مختلف مطالعه کنند. اما انتظار دارند شما بدون دانستن موضوع پاسخ هوشمندانه‌ای بدهید. آنها انتظار جادو دارند.

زمان آن رسیده است که صنعت نرم‌افزار، افسانه گورو را از بین ببرد. «گوروها» هم انسان هستند. آنها مثل بقیه ما از منطق استفاده می‌کنند و مسائل را به طور سیستماتیک تجزیه‌وتحلیل می‌کنند. از میان‌برهای ذهنی و شهود استفاده می‌کنند. بهترین برنامه‌نویسی که تابه‌حال دیده‌اید را در نظر بگیرید: یک زمانی آن شخص از الان شما در مورد نرم‌افزار کمتر می‌دانسته است. اگر کسی به نظر گورو می‌آید به‌خاطر سال‌ها تلاش برای یادگیری و بهبود فرایندهای فکری است. یک «گورو» صرفاً فرد باهوشی است با کنجکاوی بی‌پایان.

البته تفاوت عظیمی در استعداد ذاتی افراد وجود دارد. بسیاری از هکرها باهوش‌تر، بادانش بیش‌تر و بهره‌ورتر از من هستند و خواهند بود. بااین‌وجود، ازبین‌بردن افسانه گورو تأثیر مثبتی دارد. برای مثال وقتی با کسی که از من باهوش‌تر است کار می‌کنم، مطمئن می‌شوم کارهای پایه را انجام دهم و زمینه کافی را فراهم کنم تا آن شخص بتواند مهارت‌هایش را به طور مؤثر به کار گیرد. حذف افسانه گورو به معنای برداشتن مانعی ادراکی برای پیشرفت است. من به‌جای مانعی جادویی، طیف پیوسته­ای می‌بینم که می‌توانم در آن پیشرفت کنم.

در نهایت، یکی از بزرگترین موانع نرم‌افزار افراد باهوشی هستند که عمدا افسانه گورو را رواج می دهند. این کار ممکن است به دلیل غرور یا به عنوان راهبردی برای افزایش ارزش درک شده خود نزد مشتری یا کارفرما انجام شود. جالب اینجاست که این نگرش می‌تواند ارزش افراد باهوش را کمتر کند، زیرا آنها به رشد همکارانشان کمک نمی‌کنند. ما به گوروها نیاز نداریم، به متخصصانی نیاز داریم که مایل به پیشرفت متخصصان دیگر در زمینه خود باشند.

**سخت کارکردن نتیجه نمی‌دهد**

*اولو مودال*

به‌عنوان یک برنامه‌نویس، متوجه خواهید شد سخت کارکردن اغلب جواب نمی‌دهد. شاید خودتان و چند همکار را فریب دهید و باور کنید که با ساعات طولانی کارکردن، سهم بزرگی در پروژه دارید. اما حقیقت این است که با کار کمتر، شاید بتوانید به دستاورد بیشتری برسید - گاهی اوقات خیلی بیشتر. اگر سعی می‌کنید بیش از 30 ساعت در هفته متمرکز و «بهره‌ور» باشید، احتمالاً دارید خیلی سخت کار می‌کنید. برای افزایش اثربخشی و انجام کارهای بیشتر باید حجم کاری خود را کاهش دهید.

این جمله شاید غیرمنطقی و حتی بحث­برانگیز به نظر برسد، اما نتیجه مستقیم این حقیقت است که برنامه‌نویسی و توسعه نرم‌افزار به‌طورکلی، یک فرایند یادگیری مداوم هستند. هنگام کار روی یک پروژه، درک بیشتری از حوزه مسئله پیدا می‌کنید و امیدواریم روش‌های مؤثرتری برای رسیدن به هدف پیدا کنید. برای جلوگیری از هدررفتن کار، باید زمانی را برای مشاهده اثرات کاری که انجام می‌دهید، تأمل در آنچه می‌بینید و تغییر رفتار متناسب با آن اختصاص دهید.

برنامه‌نویسی حرفه‌ای معمولاً شبیه چند کیلومتر سخت دویدن نیست که هدف در انتهای جاده دیده شود. اکثر پروژه‌های نرم‌افزاری شبیه ماراتن طولانی جهت­یابی در تاریکی هستند؛ تنها با نقشه‌ای مبهم به‌عنوان راهنما. اگر فقط در یک‌جهت شروع به دویدن کنید و تا جایی که می‌توانید سریع بدوید، شاید برخی را تحت‌تأثیر قرار دهید؛ اما احتمال موفقیت کم است. باید سرعت پایداری داشته باشید و وقتی در مورد مکان فعلی و مقصد بیشتر می‌آموزید، مسیر را تغییر دهید.

علاوه بر این، شما همیشه نیاز دارید در مورد توسعه نرم‌افزار به‌طورکلی و تکنیک‌های برنامه‌نویسی به طور خاص، اطلاعات بیشتری داشته باشید. احتمالاً باید کتاب بخوانید، در کنفرانس‌ها شرکت کنید، با متخصصان دیگر ارتباط برقرار کنید، تکنیک‌های جدید پیاده‌سازی را تست کنید و در مورد ابزارهای قدرتمندی یاد بگیرید که کار شما را ساده­تر می‌کنند. باید خودتان را در زمینه‌ی تخصصی‌تان به‌روز نگه دارید، درست مثل اینکه از جراحان مغز و خلبانان انتظار می‌رود که خودشان را در زمینه‌های تخصصی‌شان به‌روز نگه دارند. شما نیاز دارید شب‌ها، آخر هفته‌ها و تعطیلات را صرف آموزش خودتان کنید؛ بنابراین، نمی‌توانید شب‌ها، آخر هفته‌ها و تعطیلات را با کار اضافی روی پروژه‌ی فعلی‌تان بگذرانید. آیا واقعاً انتظار دارید جراحان مغز 60 ساعت در هفته عمل جراحی انجام دهند یا خلبانان 60 ساعت در هفته پرواز کنند؟ البته که نه. آمادگی و آموزش بخش حیاتی حرفه‌ی آن­ها است.

روی پروژه متمرکز باشید، تا جایی که می‌توانید با یافتن راه‌حل‌های هوشمندانه در آن مشارکت کنید، مهارت‌های خود را بهبود ببخشید، در مورد کاری که انجام می‌دهید فکر کنید، رفتار خود را تطبیق دهید و از رفتارهای غیرحرفه­ای پرهیز کنید. به‌عنوان یک برنامه‌نویس حرفه‌ای، باید بدانید که تلاش برای تمرکز و «بهره­ور بودن» 60 ساعت در هفته، کار معقولی نیست. مثل یک حرفه‌ای عمل کنید: آماده شوید، تأثیر بگذارید، مشاهده کنید، تأمل کنید و تغییر دهید.

**نحوه استفاده از باگ ترکر**

*مت دور*

صرف‌نظر از اینکه آن­ها را باگ، نقص یا حتی اثرات جانبی طراحی بنامید، راه فراری از آنها وجود ندارد. دانستن نحوه ارسال یک باگ ریپورت خوب و همچنین اینکه در یک باگ ریپورت چه چیزهایی را باید جستجو کنیم، مهارت‌های کلیدی برای حفظ پیشرفت یک پروژه هستند.

یک باگ ریپورت خوب باید سه چیز را نشان دهد:

* نحوه بازتولید باگ، تاحدامکان دقیق، و اینکه هر چند وقت یکبار این باگ ظاهر می‌شود
* آنچه که باید اتفاق می‌افتاد، حداقل به نظر شما (انتظارات کاربر)
* آنچه که واقعاً اتفاق افتاده است، یا حداقل اطلاعاتی که شما ثبت کرده اید (خروجی واقعی)

میزان و کیفیت اطلاعات گزارش شده در یک باگ همان قدر در مورد گزارش‌دهنده باگ اطلاعات می‌دهد که در مورد خود باگ می‌دهد. باگ ریپورت­هایی که خشمگینانه و مختصر نوشته می‌شوند مثل «این تابع افتضاح است!»، به توسعه‌دهندگان می‌گوید که شما وقت بدی داشته‌اید، اما اطلاعات دیگر زیادی ارائه نمی‌دهد. اما یک باگ با اطلاعات کافی برای بازتولید آسان‌تر آن، نظر همه را جلب می‌کند، حتی اگر منجر به تأخیر در انتشار نسخه جدید هم بشود.

باگ­ها مثل یک گفتگو هستند، با تمام تاریخچه‌ی گذشته‌شان که در برابر همه قرار دارد. سرزنش دیگران یا انکار وجود خود باگ درست نیست. به‌جای آن، بهتر است اطلاعات بیشتری درخواست کنید یا در نظر بگیرید که چه نکته‌ای را از قلم انداخته‌اید.

تغییر وضعیت یک باگ - مثلاً از باز به بسته - یک اظهارنظر عمومی در مورد نگاه شما به آن باگ است. اختصاص‌دادن زمان برای توضیح اینکه چرا فکر می‌کنید باگ باید بسته شود، باعث می‌شود نیازی نداشته باشید ساعت‌ها مدیران و مشتریان ناراضی را در آینده توجیه کنید. تغییر اولویت یک باگ نیز یک اظهارنظر عمومی مشابه است و فقط به این دلیل که برای شما جزئی است، به این معنا نیست که مانع استفاده شخص دیگری از محصول نمی‌شود.

فیلدهای یک باگ را برای اهداف شخصی خودتان بیش از حد بارگذاری نکنید. اضافه‌کردن «VITAL:» به فیلد عنوان ممکن است برای شما مرتب‌کردن نتایج یک ریپورت را آسان‌تر کند، اما به‌مرورزمان توسط دیگران هم کپی و احتمالاً اشتباه تایپ می‌شود یا برای استفاده در ریپورت دیگری باید حذف شود. به‌جای آن از یک مقدار یا فیلد جدید استفاده کنید و مستندسازی کنید که چگونه قرار است از این فیلد استفاده شود تا دیگران مجبور به توضیح مکرر نشوند.

مطمئن شوید همه می‌دانند چگونه باگ‌هایی را پیدا کنند که تیم قرار است روی آن‌ها کار کند. معمولاً این کار با استفاده از یک جستجوی عمومی با نامی آشکار انجام می‌شود. مطمئن شوید همه از همان جستجو استفاده می‌کنند و بدون اطلاع قبلی به تیم، مبنی بر تغییر در آنچه همه روی آن کار می‌کنند، آن جستجو را به‌روز نکنید.

در نهایت به یاد داشته باشید یک باگ دقیقاً مثل یک خط کد، واحد استاندارد کار نیست و نمی‌تواند میزان تلاش لازم را به‌دقت اندازه گیرد.

**کد را با حذف بخش‌هایی از آن بهبود بخشید**

*پیت گودلیف*

کمتر، بیشتر است. این ضرب‌المثل ساده‌ای است، اما گاهی اوقات واقعاً درست است.

یکی از بهبودهایی که من طی چند هفته گذشته در کدبیس خودمان انجام دادم، حذف قسمت‌هایی از آن بود.

ما نرم‌افزار را طبق اصول XP، از جمله YAGNI (یعنی شما به آن نیاز پیدا نمی‌کنید[[7]](#footnote-7)) نوشته بودیم. باتوجه‌به طبیعت انسان، اجتناب‌ناپذیر بود در چند مورد کوتاه بیاییم.

مشاهده کردم که محصول ما برای انجام بعضی از کارهای ساده که باید تقریباً آنی باشند، زمان زیادی صرف می‌کند. دلیل من این بود که آن‌ها بیش از حد پیاده‌سازی شده بودند - مملو از زنگ‌ها و جلوه‌های اضافی بودند که لازم نبود، اما در آن زمان ایده خوبی به نظر می‌رسیدند.

پس من کد را ساده کردم، عملکرد محصول را بهبود بخشیدم و سطح آشفتگی کدهای جهانی را با حذف ویژگی‌های مشکل‌ساز از کدبیس کاهش دادم. خوشبختانه تست‌های واحد من نشان می‌دهند که هیچ‌چیز دیگری را حین این کار خراب نکرده‌ام.

یک تجربه ساده و کاملاً رضایت‌بخش بود.

چرا کدهای غیرضروری از ابتدا آنجا قرار گرفتند؟ چرا یکی از برنامه‌نویسان احساس کرد به نوشتن کد اضافی نیاز است؟ احتمالاً به دلایل زیر:

* برنامه‌نویس می‌خواست کدهای اضافی جالبی بنویسد؛ چون برایش سرگرم‌کننده بودند، نه به‌خاطر ارزش واقعی که داشتند. (نکته: وقتی کدی را اضافه کنید یا بنویسید که بدانید ارزشی را به پروژه اضافه می‌کند، نه به این دلیل که شما را سرگرم می‌کند.)
* کسی تصور داشت که ممکن است در آینده به آن نیاز باشد، پس فکر کرد بهتر است الان آن نوشت. (نکته: این YAGNI نیست. اگر در حال حاضر به آن نیاز ندارید، الان آن را ننویسید.)
* به نظر نمی‌رسید آن‌قدرها «اضافی» باشد، بنابراین پیاده‌سازی‌اش راحت‌تر از پرسیدن از مشتری بود. (نکته: نوشتن و نگهداری کد اضافی همیشه سخت‌تر است و مشتری در واقع کاملاً قابل‌دسترسی است. یک کد کوچک و اضافی به‌مرورزمان به دردسری بزرگ تبدیل می‌شود و نیاز به تعمیر دارد.)
* برنامه‌نویس نیازمندی‌های غیرواقعی ابداع کرد تا ویژگی اضافی را توجیه کند. (نکته: برنامه نویسان الزامات سیستم را تعیین نمی‌کنند، تنها مشتری نیازمندی‌ها را مشخص می‌کند.)

در حال حاضر روی چه پروژه­ای کار می‌کنید؟ آیا همه کدهایی که می‌نویسید لازم هستند؟

**نصبم کن**

*مارکوس بیکر*

من کوچک‌ترین علاقه‌ای به برنامه شما ندارم.

من در میان مشکلات زیادی گرفتار شده‌ام و لیست کارهایم بسیار طولانی است. تنها دلیلی که الان در وب‌سایت شما هستم این است که شایعه‌ای بعید شنیده‌ام مبنی بر اینکه نرم‌افزار شما تمام مشکلات من را حل خواهد کرد. امیدوارم ببخشید که درمورد آن شک دارم.

اگر مطالعات ردیابی حرکات چشم درست باشند، من تا الان عنوان را خوانده‌ام و دنبال متن آبی‌رنگ و زیرخط‌دار با عنوان «همین‌الان دانلود کنید» می‌گردم. علاوه بر این، اگر با یک مرورگر لینوکس از IP انگلستان به این صفحه رسیدم، احتمالاً نسخه لینوکس را می‌خواهم، پس لطفاً نپرسید. فرض کنیم کادر دانلود بلافاصله باز شد، من فایل را در پوشه Download قرار می‌دهم و به خواندن ادامه می‌دهم.

همه ما دائماً در حال تجزیه‌وتحلیل هزینه و فایده هر کاری که انجام می‌دهیم، هستیم. اگر پروژه شما حتی برای یک ثانیه زیر حد آستانه من باشد، آن را رها می‌کنم و به کار دیگری می‌پردازم.

اولین مانع، نصب است. فکر نمی‌کنید که این مشکل زیادی باشد؟ اکنون به پوشه Download خود بروید و نگاهی بیندازید. پر از فایل‌های tar و zip است، درست است؟ چند درصد از آنها را extract کرده‌اید؟ چندتا را نصب کرده‌اید؟ اگر شما هم مثل من باشید، تنها یک‌سوم آنها به جز اینکه فقط هارددیسک را پر کنند، فایده دیگری نیز دارند.

ممکن است دوست داشته باشم برنامه­ای سریع و آسان نصب شود، اما نمی‌خواهم بدون اجازه نصب شود. قبل از تایپ دستور install، دوست دارم دقیقاً بدانم شما فایل‌ها را کجا قرار می‌دهید. این کامپیوتر من است و دوست دارم آن را مرتب نگه دارم. همچنین می‌خواهم بتوانم برنامه شما را در لحظه‌ای که آن را نیاز ندارم، حذف کنم. اگر شک کنم این کار غیرممکن است، اصلاً آن را نصب نمی‌کنم. کامپیوتر من در حال حاضر پایدار است و می‌خواهم همین‌طور بماند.

اگر برنامه شما مبتنی بر رابط کاربری گرافیکی (GUI) است، می‌خواهم کار ساده‌ای انجام دهم و نتیجه آن را ببینم. راهنماهای گام‌به‌گام کمکی نمی‌کنند، چون کارهایی انجام می‌دهند که من آن‌ها را درک نمی‌کنم. احتمالاً می‌خواهم یک فایل بخوانم یا بنویسم. نمی‌خواهم پروژه ایجاد کنم، دایرکتوری import کنم یا آدرس ایمیلم را به شما بدهم. اگر همه چیز درست کار کند، سپس به سراغ آموزش می‌روم.

اگر نرم‌افزار شما یک کتابخانه است، من ادامه صفحه وب شما را برای یافتن راهنمای شروع سریع می‌خوانم. من معادل «Hello world» را در یک برنامه پنج‌خطی با دقیقاً خروجی توصیف شده توسط وب‌سایت شما می‌خواهم، بدون هیچ فایل یا قالب XML بزرگی، فقط یک اسکریپت. به یاد داشته باشید من فریم‌ورک رقیب شما را هم دانلود کرده‌ام، همانی که همیشه در انجمن‌ها ادعا می‌کند از شما بهتر است.

آموزشی هم وجود دارد، درست است؟ آموزشی که به زبانی صحبت کند که من درکش کنم؟

و اگر آموزش مشکل من را مطرح کند، خوشحال می‌شوم. حالا که درباره چیزهایی که می‌توانم انجام دهم می‌خوانم، جالب و حتی سرگرم‌کننده می‌شوند. به صندلی تکیه می‌دهم و چایی‌ام را می‌نوشم - گفتم که اهل انگلستان هستم؟ - و با نمونه‌های شما بازی می‌کنم و یاد می‌گیرم از خلاقیت شما استفاده کنم. اگر مشکل من را حل کند، برایتان ایمیل تشکر می‌فرستم. وقتی کرش کند، باگ ریپورت و پیشنهادهایی هم برای ویژگی‌های جدید می‌فرستم. حتی به همه دوستانم می‌گویم نرم‌افزار شما بهترین است، گرچه هرگز رقیبتان را امتحان نکردم! و همه اینها فقط به‌خاطر جلب‌اعتمادم است.

واقعاً چطور توانستم به تو شک کنم؟

**ارتباط بین فرایندها می‌تواند بر زمان پاسخ‌دهی برنامه تأثیر بگذارد**

*رندی استافورد*

زمان پاسخ‌دهی برای قابلیت استفاده نرم‌افزار بسیار مهم است. چیزی آزاردهنده‌تر از انتظارکشیدن برای پاسخ سیستم نرم‌افزاری وجود ندارد، به‌ویژه هنگامی که تعامل ما با نرم‌افزار شامل چرخه‌های تکراری محرک و پاسخ باشد. احساس می‌کنیم نرم‌افزار وقت ما را تلف می‌کند و بر بهره‌وری ما تأثیر می‌گذارد. بااین‌حال، علل ضعیف‌بودن زمان پاسخ‌دهی، به‌ویژه در برنامه‌های مدرن، کمتر شناخته شده است. بسیاری از متون مدیریت عملکرد همچنان بر ساختمان­های داده‌ها و الگوریتم‌ها تمرکز دارند، موضوعاتی که در بعضی موارد می‌توانند تفاوت ایجاد کنند؛ اما به احتمال زیاد در عملکرد برنامه‌های چندلایه‌ای مدرن نقش کمتری دارند.

وقتی مشکل عملکرد در چنین برنامه‌هایی وجود دارد، تجربه من نشان داده است که بررسی ساختمان‌های داده و الگوریتم‌ها مکان مناسبی برای یافتن بهبودها نیست. زمان پاسخ‌دهی بیشتر از همه به تعداد ارتباطات بین فرایندی راه دوری بستگی دارد که در پاسخ به یک محرک انجام می‌شوند. درحالی‌که ممکن است محدودیت‌های محلی دیگری هم وجود داشته باشند، اما تعداد ارتباطات بین فرایندی راه دور معمولاً غالب است. هر ارتباط بین فرایندی راه دوری، تأخیر غیرقابل‌چشم‌پوشی به زمان پاسخ‌دهی کلی اضافه می‌کند و این تاخیرهای جداگانه وقتی که به طور متوالی رخ می‌دهند با هم جمع می‌شوند.

یک مثال بارز آن بارگذاری موجی در یک برنامه با استفاده از نگاشت شیء-رابطه‌ای است. بارگذاری موجی به اجرای متوالی تعداد زیادی درخواست به پایگاه‌داده برای انتخاب داده‌های موردنیاز برای ساختن یک گراف از شیءها اشاره دارد. وقتی کلاینت پایگاه‌داده یک سرور، برنامه میانی باشد که یک صفحه وب را رندر می‌کند، این درخواست‌های پایگاه‌داده معمولاً به‌صورت متوالی در یک نخ واحد اجرا می‌شوند. تاخیرهای جداگانه آن‌ها با هم جمع و باعث افزایش زمان پاسخ‌دهی کلی می­شوند. حتی اگر هر درخواست پایگاه‌داده تنها 10 میلی‌ثانیه طول بکشد، یک صفحه با نیاز به 1000 درخواست (که غیرمعمول نیست) حداقل زمان پاسخ‌دهی 10 ثانیه‌ای خواهد داشت. مثال‌های دیگر شامل فراخوانی سرویس وب، درخواست‌های HTTP از مرورگر وب، فراخوانی شیء توزیع ‌شده، ارتباط دیتا-گرید و تعامل شبکه‌ داده‌ها با پروتکل‌های شبکه سفارشی است. هر چه تعداد IPCهای راه دور موردنیاز برای پاسخ به یک محرک بیشتر باشد، زمان پاسخ‌دهی بیشتر خواهد بود.

چند راهکار نسبتاً واضح و شناخته شده برای کاهش تعداد ارتباطات بین فرایندی دور به‌ازای هر محرک وجود دارد. یک راهکار اعمال اصل پارسیمونی است، به این معنی که رابط بین فرایندها به‌گونه‌ای بهینه‌سازی شود که دقیقاً داده‌های موردنیاز برای هدف موردنظر با حداقل تعامل مبادله شوند. راهکار دیگر موازی‌سازی ارتباطات بین فرایندی در صورت امکان است، به‌طوری‌که زمان پاسخ‌دهی کلی عمدتاً توسط IPC با بیشترین تأخیر تعیین شود. راهکار سوم کش کردن نتایج IPCهای قبلی است، به‌طوری‌که از IPCهای آینده از طریق دسترسی به کش محلی جلوگیری شود.

هنگام طراحی یک برنامه، به تعداد ارتباطات بین فرایندی در پاسخ به هر محرک توجه کنید. هنگام تجزیه‌وتحلیل برنامه‌هایی که دچار عملکرد ضعیف هستند، من اغلب نسبت‌های IPC به محرک را در حد هزاران به یک یافته‌ام. کاهش این نسبت، چه از طریق کش کردن یا موازی‌سازی یا تکنیک‌های دیگر، نسبت به تغییر ساختمان داده یا بهینه‌سازی الگوریتم مرتب‌سازی، فایده بسیار بیشتری خواهد داشت.

**Build را تمیز نگه دارید**

*یوهانس برادوال*

وقتی فهرستی از هشدارهای کامپایلر که به‌اندازه یک مقاله طولانی هستند را درباره کدنویسی ضعیف می‌بینیم، به خودمان می‌گوییم باید کاری در مورد آن انجام دهیم... اما الان وقت نداریم. اما وقتی یک هشدار تنها در کامپایل ظاهر می‌شود، بلافاصله آن را رفع می‌کنیم.

وقتی پروژه‌ای را از صفر شروع می‌کنیم، هیچ هشداری نیست، کد تمیز است و مشکلی وجود ندارد. اما با رشد کدبیس، اگر توجه نکنیم، می‌تواند به‌هم‌ریختگی، کدهای اضافی، هشدارها و مشکلات مختلفی روی‌هم انباشته شود. وقتی هشدارهای زیادی وجود دارد، پیداکردن هشدار مهم در بین صدها هشدار بی‌اهمیت، خیلی سخت می‌شود.

برای اینکه هشدارهای کامپایلر مفید باشند، سعی می‌کنم از سیاست تحمل صفر برای هشدارها در فرایند بیلد استفاده کنم. حتی اگر هشدار مهم نباشد، با آن سروکله می‌زنم. اگر مهم نیست؛ اما هنوز مرتبط است، آن را رفع می‌کنم. اگر کامپایلر در مورد احتمال وقوع یک Null Pointer Exception هشدار دهد، حتی اگر «بدانم» این مشکل هرگز در محیط عملیاتی رخ نمی‌دهد، علت آن را رفع می‌کنم. اگر مستندات ضمیمه‌شده در کد (Javadoc یا مشابه آن) به پارامترهایی اشاره کنند که حذف یا تغییر نام داده شده‌اند، مستندات را به‌روزرسانی می‌کنم.

اگر هشداری باشد که واقعا برایم مهم نیست و اصلا اهمیتی ندارد، از تیم می‌پرسم آیا می‌توانیم سیاست هشدارها را تغییر دهیم یا خیر. برای مثال، می‌بینم که در بیشتر موارد مستندسازی پارامترها و مقدار بازگشتی یک متد ارزش افزوده‌ای ندارد، پس اگر مستندسازی نشده‌اند، نباید هشداری باشد. یا مثلا ارتقا به نسخه جدیدتر یک زبان برنامه‌نویسی ممکن است باعث شود کدهایی که قبلا درست بودند، الان هشدار بدهند. برای مثال وقتی Java 5 برای اولین بار جنریک‌ها را معرفی کرد، تمام کدهای قدیمی که پارامتر جنریک را مشخص نکرده بودند، هشدار می‌دادند. این نوع هشدارها هستند که نمی‌خواهم در موردشان فکر کنم (حداقل فعلا نمی خواهم). داشتن مجموعه‌ای از هشدارها که با واقعیت همخوانی ندارند به نفع هیچ‌کس نیست.

با اطمینان از اینکه بیلد همیشه تمیز است، دیگر نیازی نیست هربار که یک هشدار را می‌بینم تصمیم بگیرم هشدار نامربوطی است یا خیر. نادیده‌گرفتن چیزها کاری ذهنی است و باید تلاش کنم تا جایی که می‌توانم کارهای ذهنی غیرضروری را حذف کنم. داشتن بیلد تمیز باعث می‌شود شخص دیگری هم بتواند راحت‌تر کار من را ادامه دهد. اگر هشدارها را رها کنم، شخص بعدی باید خودش تشخیص دهد چه هشدارهایی مربوط هستند و چه هشدارهایی نامربوط. یا به‌احتمال زیاد آن شخص تمام هشدارها اعم از مهم و غیرمهم را نادیده می‌گیرد.

هشدارهای حاصل از فرایند ساخت برنامه، مفید هستند. فقط نیاز است هشدارهای اضافی را حذف کنید تا بتوانید آن‌ها را متوجه شوید. منتظر یک بازبینی بزرگ نمانید، وقتی چیزی را دیدید که نمی‌خواهید ببینید، بلافاصله آن را حل کنید. باید منبع هشدار را رفع کنید، هشدار را سرکوب کنید، یا سیاست‌های هشداردهی ابزار را اصلاح کنید. تمیز نگهداشتن بیلد فقط درباره نداشتن خطاهای کامپایل یا شکست تست‌ها نیست: هشدارها هم بخش مهم و حیاتی تمیزی کد هستند.

**نحوه استفاده از ابزارهای خط فرمان را بدانید**

*کارول رابینسون*

امروزه بسیاری از ابزارهای توسعه نرم‌افزار به‌صورت محیط‌های یکپارچه توسعه (IDE) عرضه می‌شوند. Visual Studio شرکت مایکروسافت و Eclipse اوپن­سورس از جمله IDEهای محبوب هستند، هرچند IDEهای دیگری نیز وجود دارند. نکات مثبت زیادی در IDEها وجود دارد. نه‌تنها استفاده از آن­ها آسان است، بلکه برنامه‌نویس را از فکرکردن به جزئیات مربوط به فرایند ساخت معاف می‌کنند.

اما سهولت استفاده هم معایبی دارد. معمولاً زمانی که استفاده از یک ابزار آسان است، به این دلیل است که خودش تصمیم‌گیری می‌کند و بسیاری از کارها را به‌صورت خودکار و پشت پرده انجام می‌دهد؛ بنابراین اگر IDE تنها محیط برنامه‌نویسی باشد که استفاده می‌کنید، ممکن است هرگز به طور کامل درک نکنید ابزارهای شما در حقیقت چه کاری انجام می‌دهند. شما روی یک دکمه کلیک می‌کنید، اتفاق جادویی رخ می‌دهد و فایل اجرایی در پوشه پروژه ظاهر می‌شود.

با کارکردن با ابزارهای ساخت خط فرمان، درک بهتری از اینکه ابزارها هنگام ساخت پروژه چه کاری انجام می‌دهند، به دست می‌آورید. نوشتن فایل‌های make به شما کمک می‌کند تا مراحل مختلفی از جمله کامپایل کردن، ترکیب‌کردن، لینک‌کردن و غیره که منجر به تولید فایل اجرایی می‌شود را درک کنید. آزمایش گزینه‌های خط فرمان متعدد برای این ابزارها نیز تجربه آموزشی با ارزشی است. برای شروع به کار با ابزارهای ساخت خط فرمان، می‌توانید از ابزارهای اوپن سورس خط فرمان مانند GCC استفاده کنید یا ابزارهای ارائه شده با IDE اختصاصی خود را به کار ببرید. در نهایت، یک IDE خوب‌طراحی‌شده صرفاً یک رابط گرافیکی برای مجموعه‌ای از ابزارهای خط فرمان است.

علاوه بر بهبود درک شما از فرایند ساخت، برخی از وظایف به‌سادگی و کارایی بیشتری با ابزارهای خط فرمان نسبت به IDE قابل‌انجام هستند. به‌عنوان‌مثال، قابلیت‌های جستجو و جایگزینی که توسط ابزارهایی مانند grep و sed ارائه می‌شوند، اغلب قدرتمندتر از آن چیزی است که در IDEها یافت می‌شود. ابزارهای خط فرمان به طور ذاتی از اسکریپت‌نویسی پشتیبانی می‌کنند که اجازه می‌دهد وظایفی مانند تولید خودکار بیلدهای روزانه، ایجاد نسخه‌های متعدد از یک پروژه و اجرای مجموعه تست‌ها را به‌صورت خودکار انجام دهیم. در IDE انجام چنین خودکارسازی‌هایی ممکن است دشوار (یا غیرممکن) باشد، زیرا گزینه‌های بیلد معمولاً با استفاده از کادرهای گفتگویی GUI مشخص می‌شوند و فرایند ساخت با کلیک ماوس آغاز می‌شود. اگر هرگز خارج از محیط IDE کار نکنید، ممکن است متوجه نشوید که چنین وظایف خودکاری امکان‌پذیر هستند.

اما صبر کنید. آیا IDEها برای آسان‌تر کردن توسعه و بهبود بهره‌وری برنامه‌نویس نیستند؟ خب بله هدف از پیشنهاد این نیست که دیگر از IDEها استفاده نکنید، پیشنهاد این است که جزئیات IDE خود را بررسی کنید و درک کنید که IDE شما چه کاری برایتان انجام می‌دهد. بهترین راه برای انجام این کار یادگیری استفاده از ابزارهای خط فرمان است. سپس، وقتی دوباره شروع به استفاده از IDE کردید، درک بهتری از کاری که IDE برایتان انجام می‌دهد و چگونگی کنترل فرایند ساخت خواهید داشت. از طرف دیگر، بعد از یادگیری استفاده از ابزارهای خط فرمان و تجربه قدرت و انعطاف‌پذیری آن‌ها، ممکن است ترجیح دهید به‌جای IDE، از خط فرمان استفاده کنید.

**به بیش از دو زبان برنامه‌نویسی مسلط باشید**

*راسل ویندر*

روانشناسی برنامه‌نویسی: مدت‌هاست که می‌دانیم تخصص برنامه‌نویسی مستقیماً به تعداد پارادایم‌های برنامه‌نویسی مختلفی که یک برنامه‌نویس با آن‌ها احساس راحتی می‌کند، مرتبط است. یعنی نه فقط اینکه درباره‌شان می‌داند یا کمی بلد است، بلکه واقعاً می‌تواند با آن‌ها برنامه‌نویسی کند.

هر برنامه‌نویسی با یک زبان برنامه‌نویسی شروع می‌کند. آن زبان تأثیر غالبی روی نحوه فکرکردن آن برنامه‌نویس در مورد نرم‌افزار دارد. صرف‌نظر از سال‌ها تجربه‌ای که برنامه‌نویس با آن زبان به دست می‌آورد، اگر فقط همان زبان را ادامه دهد، فقط همان زبان را خواهد دانست. یک برنامه‌نویس تک‌زبانه در تفکر خود توسط همان زبان محدود می‌شود.

برنامه‌نویسی که زبان دومی یاد می‌گیرد، به چالش کشیده می‌شود، به‌ویژه اگر آن زبان مدل محاسباتی متفاوتی نسبت به زبان اول داشته باشد. سی، پاسکال، فرترن - همه مدل محاسباتی اساسی مشابهی دارند. مهاجرت از C یا Fortran به ++C یا Ada چالش‌های اساسی در نحوه رفتار برنامه‌ها ایجاد می‌کند. مهاجرت از C++ به Haskell تغییر قابل‌توجهی است و در نتیجه چالش قابل‌توجهی به شمار می‌رود. مهاجرت از C به Prolog چالش بسیار قطعی و مشخصی است.

ما می‌توانیم تعدادی از پارادایم‌های محاسباتی را نام ببریم: رویه‌ای، شیءگرا، تابعی، منطقی، جریان داده و غیره. جابه‌جایی میان این پارادایم‌ها بزرگ‌ترین چالش‌ها را ایجاد می‌کند.

چرا این چالش‌ها خوب هستند؟ این موضوع به نحوه تفکر ما در مورد پیاده‌سازی الگوریتم‌ها و اصطلاحات و الگوهای پیاده‌سازی مرتبط است. به‌ویژه، تلفیق در هسته تخصص قرار دارد، یعنی ترکیب دانش از حوزه‌های متنوع، موجب ارتقای سطح تخصص و مهارت می‌شود. اصطلاحات حل مسئله که در یک‌زبان کاربرد دارند، ممکن است در زبان دیگر کاربرد نداشته باشند. تلاش برای انتقال اصطلاحات از یک‌زبان به زبان دیگر، هم در مورد دو زبان و هم در مورد مسئله حل شده، به ما نکته‌هایی را آموزش می‌دهد.

تلفیق در استفاده از زبان‌های برنامه‌نویسی اثرات بسیار زیادی دارد. شاید مشهودترین آن، افزایش استفاده از شیوه‌های بیان اعلانی در سیستم‌های پیاده‌سازی شده با زبان‌های دستوری باشد. هر کسی که با برنامه‌نویسی تابعی آشنا باشد، حتی هنگام استفاده از زبانی مثل C می‌تواند به‌راحتی از رویکرد اعلانی استفاده کند. استفاده از رویکردهای اعلانی معمولاً منجر به برنامه‌های کوتاه‌تر و قابل‌فهم‌تر می‌شود. به‌عنوان‌مثال، ++C قطعاً این موضوع را با پشتیبانی کامل از برنامه‌نویسی generic که تقریباً نیازمند شیوه بیان اعلانی است، در نظر گرفته است.

نتیجه همه این‌ها این است که به نفع هر برنامه‌نویسی است که مهارت خوبی در برنامه‌نویسی حداقل در دو پارادایم متفاوت و ایده‌آل‌تر حداقل پنج پارادایم مذکور داشته باشد. برنامه‌نویس‌ها همیشه باید علاقه‌مند به یادگیری زبان‌های جدید، به‌ویژه از پارادایم‌های ناآشنا باشند. حتی اگر در شغل روزمره‌شان همیشه از یک‌زبان برنامه‌نویسی استفاده کنند، نباید افزایش پیچیدگی استفاده از آن زبان دست‌کم گرفته شود، هنگامی که شخص می‌تواند از دیگر پارادایم‌ها استفاده کند. کارفرمایان باید این را در نظر بگیرند و در بودجه آموزشی‌شان برای یادگیری زبان‌هایی جا باز کنند که در حال حاضر استفاده نمی‌شوند؛ اما می‌توانند پیچیدگی زبان‌های مورداستفاده را افزایش دهند.

یک دوره آموزشی یک‌هفته‌ای کافی نیست. معمولاً چند ماه تمرین و آموزش حتی پاره‌وقت نیاز است تا دانش کاربردی مناسبی از یک‌زبان به دست آید. اصطلاحات استفاده، مهم‌تر از سینتکس زبان و مدل محاسباتی هستند.

**IDE خود را بشناسید**

*هاینز کابوتز*

در دهه 1980، محیط‌های برنامه‌نویسی ما معمولاً چیزی بهتر از ویرایشگرهای متنی ساده نبودند... البته اگر خوش‌شانس بودیم و اصلاً ویرایشگری در اختیار داشتیم. برجسته‌سازی ساختار دستوری که امروزه آن را بدیهی می‌دانیم، یک امکان لوکس محسوب می‌شد که قطعاً در اختیار همه نبود. زیباسازهای کد برای فرمت‌بندی زیبای کدها معمولاً ابزارهای جداگانه‌ای بودند که باید به طور خارجی اجرا می‌شدند تا فاصله‌گذاری‌ها را اصلاح کنند. دیباگرها نیز برنامه‌های مجزایی بودند که برای گام‌به‌گام جلو رفتن در کد، با ترکیبات پیچیده‌ای از کلیدهای میان‌بر اجرا می‌شدند.

در دهه 1990، شرکت‌ها شروع به درک پتانسیل درآمدزایی کردند که می‌توانستند از طریق تجهیز برنامه‌نویسان با ابزارهای بهتر و مفیدتر آن را به دست آورند. محیط توسعه یکپارچه (IDE) ویژگی‌های ویرایشگرهای قدیمی را با یک کامپایلر، دیباگر، زیباساز کد و سایر ابزارها ترکیب کرد. در آن زمان، منوها و ماوس نیز محبوب شدند، که به این معنی بود دیگر توسعه‌دهندگان نیاز نداشتند ترکیبات پیچیده‌ای از کلیدها را برای استفاده از ویرایشگرشان یاد بگیرند. آنها می‌توانستند به‌راحتی دستور موردنظر را از منو انتخاب کنند.

در قرن 21، IDEها آنقدر رایج شده‌اند که شرکت‌هایی که می‌خواهند سهم بازار خود را در سایر حوزه‌ها افزایش دهند، آن‌ها را به صورت رایگان ارائه می‌دهند. IDEهای مدرن مجهز به مجموعه شگفت‌انگیزی از امکانات هستند. ویژگی مورد علاقه من اعمال خودکار Refactoring به ویژه Extract Method است که می‌توانم قسمتی از کد را انتخاب و به یک متد تبدیل کنم. ابزار Refactoring تمام پارامترهایی را که نیاز است به متد پاس داده شوند را شناسایی می‌کند و باعث می شود تغییر دادن کد بسیار آسان شود. IDE من حتی قسمت‌های دیگر کد را که می‌توانند توسط این متد جایگزین شوند نیز تشخیص می‌دهد و از من می‌پرسد آیا می‌خواهم آن‌ها را نیز جایگزین کنم یا خیر.

یک ویژگی شگفت‌انگیز دیگر IDEهای مدرن، توانایی اعمال دستورالعمل‌های سبک‌نویسی درون یک شرکت است. به‌عنوان‌مثال، در جاوا، برخی برنامه‌نویسان شروع به قراردادن تمام پارامترها به‌عنوان final کرده‌اند (که به نظر من، هدردادن وقت است). اما ازآنجاکه آن‌ها چنین قاعده سبک‌نویسی دارند، تنها کاری که من باید انجام دهم این است که آن را در IDE خودم تنظیم کنم: من هشداری برای هر پارامتر غیر final دریافت می‌کنم. قوانین سبک‌نویسی همچنین می‌توانند برای یافتن باگ‌های احتمالی مانند مقایسه شیء‌های باکس شده برای تساوی مرجع، مثلاً استفاده از == روی مقادیر اولیه که به شیء‌های مرجعی باکس شده‌اند، مورداستفاده قرار گیرند.

متأسفانه، IDEهای مدرن نیازی به تلاش برای یادگیری نحوه استفاده از آنها ایجاد نمی‌کنند. وقتی برای اولین بار C را در یونیکس برنامه‌نویسی می‌کردم، به خاطر منحنی یادگیری تند آن مجبور بودم زمان قابل توجهی را صرف یادگیری کار با ویرایشگر vi بکنم. این زمان صرف شده در ابتدا، طی سال ها به خوبی جبران شد. من حتی الآن پیش‌نویس این مقاله را با vi تایپ می‌کنم. IDEهای مدرن منحنی یادگیری بسیار تدریجی دارند که می‌تواند باعث شود هرگز فراتر از استفاده‌های بسیار پایه‌ای از ابزار پیشرفت نکنیم.

اولین گام من در یادگیری یک IDE حفظ‌کردن میانبرهای صفحه‌کلید است. ازآنجایی‌که انگشتانم هنگام تایپ کد روی صفحه‌کلید است، فشردن Ctrl+Shift+I برای جای‌گذاری متغیر، جریان کدنویسی را مختل نمی‌کند. اما اگر مجبور باشم برای همان کار به منوی IDE بروم و ماوس را حرکت دهم، تمرکزم از کدنویسی منحرف می‌شود. این وقفه‌ها منجر به تعویض‌های غیرضروری زمینه می‌شوند و اگر تمام کارها را به روش تنبلانه انجام دهم، بهره‌وری‌ام به‌شدت کاهش پیدا می‌کند. در مورد مهارت‌های مربوط به تایپ‌کردن هم صدق می‌کند: تایپ ده‌انگشتی یاد بگیرید. از زمانی که صرف یادگیری میانبرها و تایپ ده‌انگشتی کرده‌اید پشیمان نخواهید شد.

به‌عنوان برنامه‌نویس، به ابزارهای خط فرمان یونیکس دسترسی داریم که در طول زمان اعتبار خود را ثابت کرده‌اند و می‌توانند در دست‌کاری و تجزیه‌وتحلیل کد به ما کمک کنند. به‌عنوان‌مثال اگر هنگام بازبینی کد متوجه شوم که برنامه‌نویسان نام‌های تکراری زیادی برای کلاس‌ها استفاده کرده‌اند، می‌توانم با ترکیب ابزارهایی مانند find، sed، sort، uniq و grep، به‌راحتی آن‌ها را پیدا کنم:

find . -name "\*.java" | sed 's/.\*\///' | sort | uniq -c | grep -v "^ \*1 " | sort -r

انتظار می‌رود از ابزارهای پیشرفته‌ای مانند IDEها به طور مؤثر استفاده کنیم. پس باید زمانی را صرف یادگیری نحوه استفاده بهینه از IDE خود کنیم. همچنین نباید از قدرت ابزارهای خط فرمان غافل شد. ترکیب این دو می‌تواند بهره‌وری برنامه‌نویس را به طور چشمگیری افزایش دهد.

**محدودیت‌هایتان را بشناسید**

*گرگ کالوین*

*انسان باید محدودیت‌های خود را بشناسد.*

- درتی هری

منابع شما محدود هستند. شما فقط مقدار مشخصی وقت و پول برای انجام کارهایتان دارید، از جمله زمان و پول موردنیاز برای به‌روز نگه‌داشتن دانش، مهارت‌ها و ابزارهایتان. شما فقط می‌توانید تا حد مشخصی سخت کار کنید، سریع پیش بروید، باهوش باشید و مدت‌زمان طولانی کار کنید. ابزارهای شما فقط تا حد مشخصی قدرتمند هستند. ماشین‌های هدف شما هم فقط تا حد مشخصی قدرت دارند. پس باید محدودیت‌های منابعتان را رعایت کنید و بر اساس آن‌ها برنامه‌ریزی و عمل کنید.

برای رعایت این محدودیت‌ها چه باید کرد؟ باید خودتان، افرادتان، بودجه‌تان و محصولاتتان را بشناسید. به‌خصوص به‌عنوان یک مهندس نرم‌افزار، باید پیچیدگی مکانی و زمانی ساختمان داده‌ها و الگوریتم‌ها و همچنین معماری و ویژگی‌های عملکردی سیستم‌هایتان را بدانید. وظیفه شما ایجاد پیوند بهینه نرم‌افزار و سیستم است.

 پیچیدگی فضا و زمان به‌صورت تابع O(f(n)) نمایش داده می‌شود که در آن n برابر با اندازه ورودی است و f(n) نشان‌دهنده فضا یا زمان آسیمپتوتیک موردنیاز هنگامی است که n به سمت بی‌نهایت میل می‌کند. کلاس‌های مهم پیچیدگی برای f(n) شامل ln(n)، n، n ln(n)، ne و en هستند. همان‌طور که نمودار این توابع نشان می‌دهد، هنگامی که n بزرگ‌تر می‌شود، O(ln(n)) بسیار کوچک‌تر از O(n) و O(n ln(n)) است که آن‌ها هم بسیار کوچک‌تر از O(ne) و O(en) هستند. همان‌طور که شان پرنت می‌گوید، برای مقادیر قابل‌دستیابی n، تمام کلاس‌های پیچیدگی تقریباً ثابت، نزدیک به خطی یا نزدیک به بی‌نهایت هستند.

|  |  |  |
| --- | --- | --- |
|  | زمان دسترسی | ظرفیت |
| رجیستر | ns 1 > | b 64 |
| خط کش |  | B 64 |
| کش L1 | ns 1 | KB 64 |
| کش L2 | ns 4 | MB 8 |
| رم | ns 1 | GB 32 |
| دیسک | ms 10 | TB 10 |
| LAN | ms 20 | ZB 1 |
| اینترنت | ms 100 | PB 1 |

تحلیل پیچیدگی بر اساس یک ماشین انتزاعی اندازه‌گیری می‌شود، اما نرم‌افزارها روی ماشین‌های واقعی اجرا می‌شوند. سیستم‌های کامپیوتری مدرن به‌صورت سلسله‌مراتبی از ماشین‌های فیزیکی و مجازی سازماندهی شده‌اند که شامل مدت‌زمان‌های اجرا زبان، سیستم‌عامل‌ها، CPU ها، حافظه‌های کش، حافظه‌های دسترسی تصادفی، درایوهای دیسک و شبکه‌ها می‌شوند. این جدول محدودیت‌های زمان دسترسی تصادفی و ظرفیت ذخیره‌سازی را برای یک سرور شبکه‌ای نمونه نشان می‌دهد.

توجه داشته باشید که ظرفیت و سرعت چندین برابر با هم متفاوت است. در تمام سطوح سیستم از کش و الگوریتم‌های پیش‌بینی به طور گسترده‌ای استفاده می‌شود تا این تفاوت‌ها پنهان شوند، اما آن‌ها تنها زمانی کار می‌کنند که دسترسی قابل‌پیش‌بینی باشد. وقتی ازدست‌رفتن کش‌ها مکرر است، سیستم دچار ترش‌کردن خواهد شد. به‌عنوان‌مثال، بازرسی تصادفی هر بایت روی یک هارددیسک می‌تواند 32 سال طول بکشد. حتی بازرسی تصادفی هر بایت در رم می‌تواند 11 دقیقه طول بکشد. دسترسی تصادفی قابل‌پیش‌بینی نیست. پس چه چیزی قابل‌پیش‌بینی است؟ جواب این سؤال بستگی به سیستم دارد، اما دسترسی مجدد به آیتم‌های اخیراً استفاده شده و دسترسی ترتیبی به آیتم‌ها معمولاً سودمند است.

الگوریتم‌ها و ساختمان‌های داده‌ای، بسته به اینکه چقدر به طور مؤثر از کش‌ها استفاده می‌کنند، متفاوت هستند. به‌عنوان‌مثال:

* جستجوی خطی از پیش‌بینی به‌خوبی استفاده می‌کند، اما نیاز به O(n) مقایسه دارد.
* جستجوی باینری در یک آرایه مرتب شده فقط نیاز به O(log(n)) مقایسه دارد.

جستجو در درخت van Emde Boas با پیچیدگی O(log(n)) انجام می‌شود و مستقل از کش است.

|  |  |  |  |
| --- | --- | --- | --- |
| زمان جستجو (نانو ثانیه) | | | |
| 8 | 40 | 90 | 50 |
| 64 | 70 | 150 | 180 |
| 512 | 100 | 230 | 1200 |
| 4096 | 160 | 320 | 17000 |
|  | **vEB** | **باینری** | **خطی** |

چطور می‌توان انتخاب کرد؟ در نهایت با اندازه‌گیری. جدول زیر زمان موردنیاز برای جستجوی آرایه‌های عدد صحیح 64 بیتی را با این سه روش نشان می‌دهد. در کامپیوتر من:

* + جستجوی خطی برای آرایه‌های کوچک رقابتی است، زمان اجرای الگوریتم جستجوی خطی به‌صورت نمایی افزایش پیدا می‌کند.
  + درخت van Emde Boas، به‌خاطر الگوی دسترسی قابل پیش‌بینی‌اش، به‌وضوح برنده است و عملکرد بهتری دارد.

*شما پولتان را پرداخت می‌کنید و انتخابتان را انجام می‌دهید.*

- پانچ

**بدانید کامیت بعدی­تان چیست**

*دن برگ جانسون*

روی شانه‌های سه برنامه‌نویس زدم و پرسیدم چه‌کار می‌کنند؟ اولی پاسخ داد: «من این متدها را ریفکتور می‌کنم.» دومی پاسخ داد: «من برخی از پارامترها را به این وب اکشن اضافه می‌کنم.» سومی پاسخ داد: «من روی ویژگی‌های موردنیاز کاربر کار می‌کنم.»

شاید به نظر برسد دو نفر اول غرق در جزئیات کارشان بودند، درحالی‌که فقط سومی تمرکز بهتری داشت. اما وقتی از آنها پرسیدم که چه زمانی و چه فایل‌هایی را کامیت خواهند کرد، تصویر به طور دراماتیکی تغییر کرد. دو نفر اول به طور واضح می‌دانستند که چه فایل‌هایی درگیر خواهند بود و ظرف یک یا دو ساعت کارشان تمام می‌شد. اما برنامه‌نویس سوم جواب داد «احتمالاً ظرف چند روز آماده می‌شوم. شاید چند کلاس اضافه کنم و شاید هم آن سرویس‌ها را به نحوی تغییر دهم.»

دو نفر اول فاقد دید کلی نبودند. آنها وظایفی را انتخاب کرده بودند که فکر می‌کردند در مسیری سازنده است و ظرف چند ساعت قابل‌انجام است. وقتی این وظایف را تمام می‌کردند، ویژگی یا Refactoring جدیدی را برای کارکردن انتخاب می‌نمودند؛ بنابراین تمام کدها با هدف مشخص و قابل‌دستیابی محدودی در ذهن آنها نوشته می‌شد.

اما برنامه‌نویس سوم نتوانسته بود مسئله را تجزیه کند و روی همه جنبه‌های آن به طور هم‌زمان کار می‌کرد. او هیچ تصوری از چیزی که لازم بود انجام دهد نداشت و در واقع به‌صورت حدسی برنامه‌نویسی می‌کرد تا شاید به نقطه‌ای برسد که بتواند کامیت کند. احتمالاً کدهایی که در ابتدای این جلسه طولانی نوشته شده بودند، برای راه‌حل نهایی که در پایان پیدا شده است، مناسب نبودند.

اگر وظایف دو برنامه‌نویس اول بیش از دو ساعت طول می‌کشید، احتمالاً چه می‌کردند؟ بعد از اینکه متوجه می‌شدند وظیفه بیش از حد بزرگی را انتخاب کرده‌اند، احتمالاً تغییرات خود را کنار می‌گذاشتند، وظایف کوچک‌تری تعریف می‌کردند و از اول شروع می‌کردند. ادامه کار بدون تمرکز، منجر به ورود کدهای حدسی به ریپازیتوری می‌شد. به‌جای آن، تغییرات کنار گذاشته می‌شد؛ اما بینش به‌دست‌آمده حفظ می‌شد.

اما برنامه‌نویس سوم احتمالاً به حدس‌زدن ادامه می‌داد و با اصرار تلاش می‌کرد تغییراتش را به چیزی تبدیل کند که بتواند آن‌ها را کامیت کند و در نهایت، فکر می‌کند نمی‌توان تغییرات کدی را که انجام شده‌اند کنار گذاشت، چون کار بیهوده‌ای خواهد بود! متأسفانه، اصرار بر کامیت کردن تغییرات منجر به ورود کدهای عجیب و بی‌هدف به کدبیس می‌شود.

در نهایت، حتی برنامه‌نویسانی که بر کامیت کردن متمرکز هستند هم ممکن است نتوانند چیز مفیدی پیدا کنند که فکر کنند ظرف دو ساعت قابل‌انجام است. در این صورت، مستقیماً وارد حالت حدسی می‌شوند، با کد بازی می‌کنند و البته هر زمان بینشی به آن‌ها کمک کرد مسیر را پیدا کنند، تغییرات را دور می‌ریزند. حتی این جلسات به‌ظاهر بی‌ساختار هک‌کردن هم هدفی دارند: یادگیری در مورد کد برای توانایی تعریف یک وظیفه که گامی سازنده به جلو باشد.

بدانید کامیت بعدی­تان چیست. اگر نمی­توانید آن را تمام کنید، تغییرات را کنار بگذارید، سپس وظیفه جدیدی را با بینش به‌دست‌آمده تعریف کنید که به آن اعتقاد داشته باشید. هر زمان لازم بود آزمایش‌های حدسی انجام دهید، اما اجازه ندهید بدون اینکه متوجه شوید وارد حالت حدس و گمان شوید. حدس و گمان‌ها را در ریپازیتوری کامیت نکنید.

**داده‌های بزرگ و به‌هم‌پیوسته متعلق به یک پایگاه‌داده هستند**

*دیومیدیس اسپینلیس*

اگر برنامه شما قرار است مجموعه بزرگ، پایدار و مرتبطی از عناصر داده­ای را مدیریت کند، در استفاده از یک پایگاه‌داده رابطه‌ای تردید نکنید. درگذشته RDBMSها گران‌قیمت، کمیاب، پیچیده و دست‌وپاگیر بودند؛ اما دیگر این‌طور نیست. امروزه سیستم‌های RDBMS به‌راحتی قابل‌دسترس هستند. احتمالاً سیستمی که استفاده می‌کنید از قبل یک یا دو عدد از آن‌ها را نصب شده دارد. برخی RDBMSهای قدرتمند مثل MySQL و PostgreSQL به‌صورت اوپن­سورس در دسترس هستند، پس برای خرید نیازی نیست هزینه­ای پرداخت کنید. بهتر از آن، RDBMSهای جاسازی‌شده می‌توانند به‌صورت کتابخانه، مستقیماً در برنامه متصل شوند و نیاز به راه‌اندازی و مدیریت چندانی ندارند. دو نمونه اوپن­سورس معروف عبارت‌اند از SQLite و HSQLDB. این سیستم‌ها بسیار کارآمد هستند.

اگر حجم داده‌های برنامه بزرگ‌تر از حافظه RAM سیستم باشد، یک جدول RDBMS شاخص‌گذاری شده، نسبت به انواع مپ در کتابخانه‌ها که منجر به ترش‌کردن صفحات حافظه مجازی می‌شوند، سرعتی چندین برابر بیشتر خواهد داشت. پایگاه‌داده‌های مدرن به‌راحتی می‌توانند با نیازهای شما رشد کنند. با مراقبت، زمانی که نیاز است، می‌توان یک پایگاه‌داده جاسازی‌شده را به پایگاه‌داده بزرگ‌تری تبدیل کرد. بعداً می‌توان از یک سیستم رایگان و اوپن­سورس به سیستم انحصاری پشتیبانی­شده بهتر یا قدرتمندتری سویچ کرد.

یک‌بار که SQL را یاد گرفتید، نوشتن برنامه‌های متمرکز بر پایگاه‌داده لذت‌بخش می‌شود. پس از ذخیره درست داده‌های نرمال شده در پایگاه‌داده، استخراج داده‌ها به‌صورت کارآمد با یک عبارت SQL خوانا آسان است؛ نیازی به نوشتن کد پیچیده نیست. به همین ترتیب، یک دستور SQL می‌تواند تغییرات پیچیده‌ای روی داده‌ها انجام دهد. برای تغییرات یک‌باره - مثلاً تغییر در نحوه سازمان‌دهی داده‌های پایدار - حتی نیازی به نوشتن کد نیست: کافی است اینترفیس SQL پایگاه‌داده را اجرا کنید. همین اینترفیس، بدون نیاز به چرخه کامپایل-ویرایش زبان برنامه‌نویسی، اجازه می‌دهد کوئری‌ها را آزمایش کرد.

مزیت دیگر استفاده از RDBMS برای کدنویسی، مربوط به مدیریت روابط بین عناصر داده‌ای است. می‌توانید محدودیت‌های یکپارچگی داده‌ها را به‌صورت توصیفی مشخص کنید و از ریسک اشاره‌گرهای آویزان که در صورت فراموش‌کردن به‌روزرسانی داده‌ها در موردهای لبه‌ای اتفاق می‌افتد، جلوگیری کنید. به‌عنوان‌مثال می‌توانید مشخص کنید اگر یک کاربر حذف شد، پیام‌های ارسال شده توسط او هم باید حذف شوند.

همچنین می‌توانید با ایجاد یک شاخص، در هر زمان که بخواهید لینک‌های کارآمدی بین عناصر ذخیره شده در پایگاه‌داده ایجاد کنید. نیازی به انجام Refactoring پرهزینه و گسترده روی فیلدهای کلاس‌ها نیست. علاوه بر این، کدنویسی بر پایه پایگاه‌داده اجازه دسترسی امن چند برنامه به داده‌های شما را می‌دهد. این کار ارتقای برنامه را برای استفاده هم‌زمان و همچنین نوشتن هر بخش برنامه با زبان و پلتفرم مناسب آسان می‌کند. به‌عنوان‌مثال می‌توانید بک‌اند XML یک برنامه مبتنی بر وب را در Java، اسکریپت‌های حسابرسی را در Ruby و رابط گرافیکی را در Processing بنویسید.[[8]](#footnote-8)

در نهایت این نکته را به یاد داشته باشید که RDBMS تلاش زیادی برای بهینه‌سازی دستورات SQL شما انجام می‌دهد تا بتوانید به‌جای تنظیم الگوریتمی، روی عملکرد برنامه تمرکز کنید. پایگاه‌داده‌های پیشرفته حتی از پردازنده‌های چندهسته‌ای هم بهره می‌برند و البته با بهبود فناوری، عملکرد برنامه شما نیز بهبود پیدا می‌کند.

**زبان‌های خارجی را یاد بگیرید**

*کلاوس مارکوارت*

برنامه‌نویس‌ها نیازمند ارتباط‌گیری زیادی هستند.

دوره‌هایی در زندگی یک برنامه‌نویس وجود دارد که بیشتر ارتباطات به نظر می‌رسد با کامپیوتر است - به طور دقیق‌تر، با برنامه‌هایی که روی آن کامپیوتر اجرا می‌شوند. این ارتباط در مورد بیان ایده‌ها به شکلی است که توسط ماشین قابل خواندن باشد. برنامه‌ها ایده‌هایی هستند که تقریباً بدون کمک هیچ ماده فیزیکی، به واقعیت تبدیل می‌شوند.

برنامه‌نویس‌ها باید به زبان ماشین، چه واقعی و چه مجازی، و انتزاع‌هایی مسلط باشند که از طریق ابزارهای توسعه می‌توانند به آن زبان مرتبط شوند. یادگیری انتزاع‌های مختلف مهم است، در غیر این صورت برخی ایده‌ها بسیار سخت بیان می‌شوند. برنامه‌نویس‌های خوب باید بتوانند از روال روزمره خود خارج شوند و زبان­های دیگری را بدانند که برای اهداف دیگر مفید هستند. همیشه زمانی فرامی‌رسد که این کار نتیجه مثبتی می‌دهد.

علاوه بر ارتباط با ماشین‌ها، برنامه‌نویس‌ها نیازمند ارتباط با همکاران خود نیز هستند. پروژه‌های بزرگ امروزی بیش از صرف هنر کاربردی برنامه‌نویسی، مربوط به تلاش‌های اجتماعی هستند. درک و بیان بیش از آنچه انتزاع‌های قابل خواندن توسط ماشین ارائه می‌دهند، مهم است. بیشتر برنامه‌نویسان خوبی که می‌شناسم در زبان مادری خود معمولاً در زبان‌های دیگر نیز بسیار روان هستند. این فقط در مورد ارتباط با دیگران نیست: به‌خوبی صحبت‌کردن به یک‌زبان همچنین منجر به روشنفکری می‌شود که در هنگام انتزاع مسائل ضروری است و برنامه‌نویسی نیز به این موضوع ربط دارد.

فراتر از ارتباط با ماشین، خود و همکاران، یک پروژه ذی‌نفعان متعددی دارد که اغلب سابقه فنی متفاوت دارند یا اصلاً سابقه­ای ندارند و در بخش‌های تست، کیفیت و انتشار؛ بازاریابی و فروش فعالیت می‌کنند یا کاربران نهایی در دفتر (یا فروشگاه یا خانه) هستند. شما نیاز دارید آن­ها و نگرانی‌هایشان را درک کنید. این کار تقریباً غیرممکن است اگر نتوانید به زبان آن­ها - زبان دنیا و حوزه‌شان - صحبت کنید. درحالی‌که ممکن است فکر کنید مکالمه با آن­ها خوب پیش رفته، احتمالاً آن­ها همچون فکری نمی­کنند.

اگر با حسابداران صحبت می‌کنید، نیاز به دانش‌پایه‌ای از حسابداری مرکز هزینه، سرمایه ثابت و سرمایه به کار گرفته شده و غیره دارید. اگر با بازاریابان یا وکلا صحبت می‌کنید، برخی از اصطلاحات و زبان آ­نها (و در نتیجه ذهن آن­ها) باید برای شما آشنا باشد. در پروژه باید کسی به همه این زبان‌های مخصوص به هر حوزه مسلط باشد - ایده‌آل این است که برنامه‌نویس‌ها این کار را انجام دهند. در نهایت، برنامه‌نویس‌ها مسئول تبدیل ایده‌ها به واقعیت از طریق کامپیوتر هستند.

و البته زندگی فراتر از پروژه‌های نرم‌افزاری است. همان‌طور که شارلمانی گفته است، دانستن زبانی دیگر مانند داشتن روحی دیگر است. برای ارتباطات فراتر از صنعت نرم‌افزار، دانستن زبان‌های خارجی، اینکه چه زمانی باید به‌جای حرف‌زدن گوش داد و اینکه بیشتر زبان بدون کلمات است، مفید خواهد بود.

*در مورد هر چیزی که نمی‌توان صحبت کرد، باید در موردش سکوت کرد.*

- لودویگ ویتگنشتاین

**یاد بگیرید برآورد کنید**

*جیووانی آسپرونی*

به‌عنوان یک برنامه‌نویس، باید بتوانید برای مدیران، همکاران و کاربرانتان، برآوردهایی از زمان و تلاش موردنیاز برای انجام وظایف محوله ارائه دهید تا آنها تصور نسبتاً دقیقی از زمان، هزینه‌ها، فناوری و سایر منابع موردنیاز برای رسیدن به اهدافشان داشته باشند.

برای اینکه بتوانید خوب برآورد کنید، مشخصاً مهم است که برخی تکنیک‌های برآورد را یاد بگیرید. اما قبل از هر چیز، درک اینکه برآوردها چه هستند و باید برای چه مورداستفاده قرار بگیرند، ضروری است. عجیب است؛ اما بسیاری از توسعه‌دهندگان و مدیران واقعاً نمی‌دانند که برآوردها برای چه هستند.

پس مهم است که ابتدا ماهیت و کاربرد برآوردها را بفهمیم و سپس تکنیک‌های مختلف برآورد را یاد بگیریم تا بتوانیم برآوردهای دقیق و مفیدی ارائه کنیم.

این گفتگو بین مدیر پروژه و برنامه‌نویس گفتگوی غیرمعمولی نیست:

مدیر پروژه: می‌توانی برآوردی از زمان لازم برای توسعه ویژگی xyz به من بدهی؟

برنامه‌نویس: یک ماه.

مدیر پروژه: یک ماه خیلی زیاد است! فقط یک هفته زمان داریم.

برنامه‌نویس: من حداقل سه هفته نیاز دارم.

مدیر پروژه: ولی من حداکثر می‌توانم دو هفته به شما وقت بدهم.

برنامه‌نویس: قبول!

در نهایت، برنامه‌نویس «برآوردی» ارائه می‌دهد که با آنچه برای مدیر قابل‌قبول است، تطابق دارد. این برآورد واقعی برنامه‌نویس نیست و صرفاً برای رضایت مدیر پروژه تغییر کرده است؛ اما ازآنجایی‌که به‌عنوان برآورد برنامه‌نویس در نظر گرفته می‌شود، مدیر او را مسئول آن برآورد می‌داند. برای درک اینکه در این گفتگو چه اشکالی وجود دارد، نیاز به سه تعریف داریم - برآورد، هدف و تعهد:

* برآورد، یک محاسبه یا قضاوت تقریبی از ارزش، تعداد، کمیت یا میزان چیزی است. این تعریف به معنای آن است که برآورد باید بر اساس داده‌های عینی و تجربیات قبلی محاسبه شود - آرزوها و خواسته‌ها باید هنگام محاسبه آن نادیده گرفته شوند. همچنین به معنای آن است که برآورد نمی‌تواند دقیق باشد، مثلاً نمی‌توان گفت یک تسک توسعه 234.14 روز طول می‌کشد.
* هدف، بیان یک خواسته کسب‌وکاری مطلوب است، مثلاً «سیستم باید حداقل 400 کاربر همزمان را پشتیبانی کند.»
* تعهد، وعده‌ای برای ارائه عملکرد مشخصی با سطح کیفیت معین تا تاریخ یا رویداد خاصی است. مثال آن می‌تواند این باشد: «قابلیت جستجو در نسخه بعدی محصول در دسترس خواهد بود.»

برآوردها، اهداف و تعهدات مستقل از هم هستند، اما اهداف و تعهدات باید بر اساس برآوردهای صحیح باشند. همان‌طور که استیو مک‌کانل می‌گوید: «هدف اصلی برآورد نرم‌افزاری پیش‌بینی نتیجه پروژه نیست؛ بلکه تعیین این است که آیا اهداف پروژه به‌اندازه کافی واقع‌بینانه هستند که اجازه کنترل پروژه برای رسیدن به آن‌ها را بدهند یا خیر.» بنابراین هدف برآورد این است که مدیریت و برنامه‌ریزی صحیح پروژه را ممکن سازد، تا ذی‌نفعان پروژه بتوانند بر اساس اهداف واقع‌بینانه تعهداتی را ارائه کنند.

آنچه مدیر در گفتگوی قبلی واقعاً از برنامه‌نویس خواسته بود، ارائه تعهدی بر اساس هدفی ذهنی بود که مدیر در ذهن داشت، نه ارائه برآورد. بار دیگری که از شما خواسته شد برآوردی ارائه دهید، مطمئن شوید همه کسانی که در آن پروژه دخیل هستند می‌دانند در مورد چه چیزی صحبت می‌کنند؛ با این کار پروژه‌های شما شانس بهتری برای موفقیت خواهند داشت. حال وقت آن است چند تکنیک یاد بگیریم...

**«Hello, World» نوشتن را یاد بگیرید**

*توماس گِست*

پل لی که با نام کاربری «لیپ» و بیشتر با نام مستعار «هاپی» شناخته می‌شد، در شرکت ما به‌عنوان متخصص مسائل برنامه‌نویسی شهرت داشت. به کمک نیاز داشتم پس به سمت میز هاپی رفتم و پرسیدم آیا می‌تواند به کد من نگاهی بیندازد یا نه.

هاپی گفت: «باشه، صندلی بیاور.» مراقب بودم که قوطی‌های خالی کولای او را که به شکل هرمی در پشتش چیده شده بود نیندازم.

پرسید: «چه کدی؟»

گفتم: «کدی در یک تابع از یک فایل.»

هاپی گفت: «خب این تابع رو بررسی کنیم.» او یک کپی از کتاب K&R[[9]](#footnote-9) را کنار زد و صفحه‌کلیدش را جلوی من گذاشت.

پرسیدم: «IDE کجاست؟» ظاهراً هاپی هیچ IDE در حال اجرایی نداشت، فقط یک ادیتور بود که من نمی‌توانستم با آن کار کنم. او دوباره صفحه‌کلید را جلوی خودش گذاشت. چند کلید فشرد بعد فایل باز شد - فایل بزرگی بود - و تابع را دیدیم - تابع بزرگی بود. او صفحه را به پایین کشید تا به بلاک کد شرطی برسد که می‌خواستم درباره‌اش بپرسم.

پرسیدم: «اگر x منفی باشد، این شرط چه کاری انجام می‌دهد؟ مسلماً اشتباه است.»

تمام صبح سعی کرده بودم راهی پیدا کنم که x را منفی کنم، اما آن تابع بزرگ در آن فایل بزرگ بخشی از یک پروژه بزرگ بود و چرخه مجدد کامپایل کردن و سپس اجرای دوباره، مرا خسته کرده بود. آیا یک متخصص مثل هاپی نمی‌توانست جواب را به من بگوید؟

هاپی اعتراف کرد که مطمئن نیست. برایم جای تعجب داشت که او سمت کتاب K&R نرفت. در عوض، بلاک کد را در بافر ویرایشگر جدید کپی کرد، آن را دوباره فرمت کرد و با یک تابع رپ کرد. چند لحظه بعد تابع main را کدنویسی کرده بود. این تابع به طور مداوم تکرار می‌شد، از کاربر مقادیر ورودی را می‌گرفت، مقادیر ورودی را به تابع می‌فرستاد و نتیجه را چاپ می‌کرد. او بافر را به‌عنوان فایل جدید tryit.c ذخیره کرد. اگرچه شاید نه به سرعت او اما من هم می‌توانستم این کار را انجام دهم. اما گام بعدی او ساده و در آن زمان برای من کاملاً بیگانه بود:

$ cc tryit.c && ./a.out

ببینید برنامه واقعی او که فقط چند دقیقه قبل طراحی شده بود، اکنون در حال اجرا بود. ما چند مقدار را امتحان کردیم و ظن من را تأیید کردیم (پس در مورد چیزی حق با من بود!) سپس او بخشی از کتاب K&R را که به این موضوع مربوط بود، بررسی کرد. من از هاپی تشکر کردم و آنجا را ترک کردم و دوباره دقت کردم هرم قوطی‌های کولای او را خراب نکنم.

وقتی به میز خودم برگشتم، IDE خود را بستم. آن‌قدر عادت کرده بودم که روی یک پروژه بزرگ درون یک محصول بزرگ کار کنم که فکر می‌کردم این کاری است که باید انجام دهم. یک کامپیوتر چندمنظوره هم می‌تواند وظایف کوچکی انجام دهد. یک ادیتور متن باز کردم و شروع کردم به تایپ‌کردن:

#include <stdio.h>

int main()

{

printf("Hello, World\n");

return 0;

}

**اجازه دهید پروژه شما خودش صحبت کند**

*دانیل لیندر*

به نظر می‌رسد پروژه شما دارای یک زیرساخت مناسب کنترل نسخه و یکپارچه‌سازی مداوم است که از طریق تست‌های خودکار صحت کد را تأیید می‌کند. این فوق‌العاده است.

شما می‌توانید ابزارهای تحلیل کد استاتیک را نیز برای جمع‌آوری معیارهایی از کد مانند درصد پوشش تست اضافه کنید. این اطلاعات بازخورد مفیدی در مورد کیفیت و تکامل کد در طول زمان فراهم می‌کنند. سپس می‌توانید خط‌قرمزهایی مانند حداقل میزان پوشش تست مشخص کنید. فرض کنید شما با 20٪ پوشش تست شروع کرده‌اید و نمی‌خواهید زیر 15٪ بیایید. یکپارچه‌سازی مداوم به شما کمک می‌کند تا از این اعداد پیروی کنید، اما همچنان باید به طور منظم آن را بررسی کنید. تصور کنید بتوانید این وظیفه را به خود پروژه واگذار کنید و اگر این معیارها نقض شدند، اجازه دهید خود سیستم به شما هشدار دهد.

شما نیاز دارید که به پروژه خود صدایی بدهید. این کار می‌تواند با ارسال ایمیل یا پیام فوری به توسعه‌دهندگان برای اطلاع از آخرین تغییرات و وضعیت شاخص‌های کلیدی مانند کاهش یا بهبود درصد پوشش تست، پیچیدگی سیکلوماتیک و غیره انجام شود. اما روش مؤثرتر این است که به پروژه در محیط کاری با استفاده از یک دستگاه بازخورد شدید (XFD) عینیت ببخشید.

ایده XFDها این است که یک دستگاه فیزیکی مانند یک چراغ، فواره قابل‌حمل، ربات اسباب‌بازی یا حتی یک موشک‌انداز USB را بر اساس نتایج تجزیه‌وتحلیل خودکار وضعیت پروژه کنترل کنید. هر زمان شاخص‌ها از محدوده مجاز تعیین‌شده خارج شوند، دستگاه وضعیت خود را به صورتی آشکار تغییر می‌دهد. به‌عنوان‌مثال برای یک چراغ رومیزی، روشن می‌شود و به طور واضح قابل‌مشاهده است. حتی اگر توسعه‌دهندگان به سرعت محل کار خود را ترک کنند، نمی‌توانند پیام را از دست بدهند.

بسته به نوع دستگاه بازخورد شدیدی که استفاده می‌کنید، می‌توانید صدای شکستن ساخت را بشنوید، سیگنال‌های هشدار قرمز را در کدتان ببینید یا حتی بوی کدهای بدتان را استشمام کنید. اگر در یک تیم توزیع‌شده کار می‌کنید می‌توانید این XFDها را در مکان‌های مختلف قرار دهید تا همه بتوانند وضعیت پروژه را مشاهده کنند. برای مثال، می‌توانید یک چراغ‌راهنمایی در دفتر مدیر پروژه بگذارید تا وضعیت کلی پروژه را نمایش دهد. مدیر پروژه شما از این کار قدردانی خواهد کرد.

انتخاب نوع XFD بستگی به خلاقیت و فرهنگ سازمانی شما دارد. اگر فضای کاری شما غیررسمی‌تر است، می‌توانید از اسباب‌بازی‌های کنترل از راه دور برای ماسکوت تیم استفاده کنید. اما اگر محیط کاری رسمی‌تر است، چراغ‌های طراحی‌شده انتخاب بهتری است. برای ایده‌های بیشتر اینترنت را بگردید. هر چیزی با یک پریز برق یا کنترل از راه دور می‌تواند به‌عنوان یک دستگاه بازخورد شدید استفاده شود.

XFD مانند جعبه صدای پروژه عمل می‌کند و وضعیت آن را برای توسعه‌دهندگان قابل تجسم کنید. تیم می‌تواند قوانینی تعیین کند که XFD بر اساس آن‌ها عملکرد تیم را تحسین یا از آن شکایت کند. با اضافه‌کردن سنتز گفتار و بلندگو، XFD می‌تواند خودش صحبت کند و بازخورد بدهد.

**لینکر یک برنامه جادویی نیست**

*والتر برایت*

به نظر می‌رسد بسیاری از برنامه‌نویسان درک درستی از مراحل تبدیل سورس کد به فایل اجرایی ایستا و لینک‌شده در زبان‌های کامپایلی ندارند. این مشکل پیش از نوشتن این متن دوباره برای خود من هم پیش آمد. تصور برنامه‌نویسان از این فرایند اغلب چنین است:

1. ویرایش کد منبع
2. کامپایل کد منبع به فایل‌های شی
3. رخ‌دادن اتفاقی جادویی
4. اجرای برنامه

البته مرحله سوم همان مرحله لینک‌کردن است. چرا این‌طور فکر می‌کنم؟ زیرا سال‌هاست که پشتیبانی فنی انجام می‌دهم و بارها با سؤال‌های زیر مواجه شده‌ام:

1. لینکر می‌گوید def چند بار تعریف شده است.
2. لینکر می‌گوید abc یک نماد مبهم است.
3. چرا فایل اجرایی این‌قدر بزرگ است؟

این سؤال‌های مربوط به لینکر معمولاً با عباراتی مثل «به نظر می‌رسد» و «به‌نوعی» همراه هستند که نشان‌دهنده سردرگمی و بی‌اطلاعی کامل شخص پرسشگر است. استفاده از این عبارات نشان می‌دهد فرایند لینک‌کردن برای برنامه‌نویسان مثل یک جعبه سیاه جادویی به نظر می‌رسد که فقط افراد خاصی مثل جادوگران و افسونگران می‌توانند آن را درک کنند. اما در مورد فرایند کامپایل کردن چنین عبارات سردرگم کننده‌ای به کار نمی‌رود که نشان می‌دهد برنامه‌نویسان معمولاً کارکرد کامپایلر را تا حدودی درک می‌کنند.

لینکر یک برنامه ساده، معمولی و بدون پیچیدگی است. تنها کاری که انجام می‌دهد اتصال بخش‌های کد و داده‌های فایل‌های شیء، ارتباط‌دادن مراجع به نمادها با تعاریف آن‌ها، بیرون‌کشیدن نمادهای مبهم از کتابخانه و نوشتن فایل اجرایی است. هیچ جادو یا افسونی در کار نیست! سختی نوشتن لینکر معمولاً به‌خاطر رمزگشایی و تولید فرمت‌های فایل پیچیده است، اما این موضوع ماهیت اصلی لینکر را تغییر نمی‌دهد.

بنابراین، فرض کنید لینکر اعلام می‌کند که def چندین بار تعریف شده است. بسیاری از زبان‌های برنامه‌نویسی مثل C، ++C و D هم اعلان و هم تعریف دارند. اعلان‌ها معمولاً در فایل‌های هدر قرار می‌گیرند، مانند:

extern int iii;

که یک مرجع خارجی به نماد iii ایجاد می‌کند. از سوی دیگر، یک تعریف، فضایی برای نماد در حافظه اختصاص می‌دهد و معمولاً در فایل پیاده‌سازی ظاهر می‌شود:

int iii = 3;

هر نماد چند تعریف می‌تواند داشته باشد؟ مثل فیلم هایلندر، تنها می‌تواند یک تعریف داشته باشد. پس اگر تعریف iii در بیش از یک فایل پیاده‌سازی تکرار شود چه می‌شود؟

// File a.c

int iii = 3;

// File b.c

double iii(int x) { return 3.7; }

نه‌تنها تعریف باید یکتا باشد، بلکه یک نماد حتماً باید حداقل یک تعریف داشته باشد. اگر iii فقط به‌صورت اعلان ظاهر شود و هرگز تعریف نشود، لینکر از iii به‌عنوان یک نماد مبهم شکایت می‌کند.

برای بررسی دلیل اندازه بزرگ فایل اجرایی هم می‌توان از فایل Map تولید شده توسط لینکر استفاده کرد که نشان‌دهنده ماژول‌ها و اندازه آن‌هاست. فایل Map فقط لیستی از همه نمادهای موجود در فایل اجرایی به همراه آدرس آن‌هاست. این فایل به شما می‌گوید کدام ماژول‌ها از کتابخانه لینک شده‌اند و اندازه هرکدام از آن‌ها چقدر است. حالا می‌توانید ببینید مشکل از کجاست. اغلب، ماژول‌هایی از کتابخانه وجود دارند که نمی‌دانید چرا لینک شده‌اند. برای پیداکردن علت، موقتاً آن ماژول مشکوک را از کتابخانه حذف کنید و دوباره لینک کنید. خطای نماد تعریف‌نشده، نشان می‌دهد چه کسی به آن ماژول ارجاع داده است.

اگرچه گاهی اوقات دلیل دریافت پیام خاصی از لینکر، بلافاصله مشخص نیست، اما هیچ چیز جادویی در لینکر وجود ندارد. مکانیسم آن ساده است؛ فقط باید جزئیات را متوجه شوید.

**طول عمر راه‌حل‌های موقت**

*کلاوس مارکوارت*

معمولاً دلیل ایجاد راه‌حل‌های موقت این است که مشکل فوری‌ای وجود دارد که باید حل شود. ممکن است این مشکل داخلی در تیم توسعه باشد، مثلاً ابزاری که یک نقص در زنجیره ابزاری را برطرف می­کند. یا ممکن است خارجی باشد و کاربران نهایی آن را ببینند، مثل راه‌حلی موقت برای کارکرد ناقص.

در بیشتر سیستم‌ها و تیم‌ها، نرم‌افزارهایی یافت می‌شود که تا حدی از سیستم جدا هستند، قرار است در آینده تغییر کنند، استانداردها و دستورالعمل‌های کدنویسی بقیه کد را رعایت نمی‌کنند. توسعه‌دهندگان اغلب از این نرم‌افزارها شکایت دارند. دلایل ایجاد آن‌ها متنوع است، اما کلید موفقیت یک راه‌حل موقت، ساده است: مفیدبودن آن. اگر راه‌حلی موقت، نیازی را برآورده کند، موفق خواهد بود. اما اگر بی‌فایده باشد، زود حذف می‌شود.

با اینکه راه‌حل‌های موقت قرار است موقتی باشند، اما معمولاً دوام و پایداری زیادی پیدا می‌کنند. چون این راه‌حل‌ها برای یک مشکل فوری مفید بوده‌اند و پذیرفته شده‌اند، انگیزه‌ای برای تغییر آن‌ها وجود ندارد. هنگام تصمیم‌گیری در مورد اولویت‌بندی کارها، ادغام صحیح یک راه‌حل موقت اغلب در اولویت پایین‌تری قرار می‌گیرد. چرا؟ چون آن راه‌حل در حال حاضر مفید است و کسی مشکلی با آن ندارد. تنها مسئله‌اش رعایت‌نکردن استانداردهاست که چندان مهم تلقی نمی‌شود.

بنابراین، راه‌حل موقت برای همیشه بدون تغییر باقی می‌ماند، برای همیشه.

حتی اگر مشکلی هم پیش بیاید، به‌جای اصلاح ریشه‌ای، یک آپدیت موقت دیگر اعمال می‌شود. پس باید چه کرد؟ باید مراقب بود که راه‌حل‌های موقت واقعاً موقت بمانند و سریعاً جایگزین شوند. وگرنه برای همیشه در کد باقی خواهند ماند.

آیا این یک مشکل است؟

پاسخ این سؤال بستگی به پروژه و سهم شخصی شما در استانداردهای کد تولیدی دارد. هنگامی‌که سیستم شامل راه‌حل‌های موقت زیادی است، آنتروپی یا پیچیدگی داخلی آن افزایش و قابلیت نگهداری آن کاهش می‌یابد. بااین‌حال، احتمالاً این اولین سؤالی نیست که باید پرسیده شود. به یاد داشته باشید که ما در مورد یک راه‌حل صحبت می‌کنیم. ممکن است راه‌حل موردعلاقه شما نباشد - بعید است راه‌حل موردعلاقه کسی باشد - اما انگیزه برای بازنگری این راه‌حل ضعیف است.

اگر مشکلی مشاهده کردیم، چه کارهایی می‌توانیم انجام دهیم؟

1. از ابتدا از ایجاد یک راه‌حل موقت جلوگیری کنیم.
2. نیروهایی را که بر تصمیم‌گیری مدیر پروژه تأثیر می‌گذارند، تغییر دهیم.
3. وضعیت فعلی را حفظ کنیم.

بیایید این گزینه‌ها را دقیق‌تر بررسی کنیم:

1. اجتناب از راه‌حل‌های موقت همیشه امکان‌پذیر نیست. گاهی مشکل واقعی وجود دارد و استانداردها بسیار محدود‌کننده‌اند. شاید بتوان استانداردها را تغییر داد؛ اما این تغییرات به‌موقع برای حل مشکل فعلی اعمال نمی‌شوند.
2. مقاومت در برابر تغییر، ریشه در فرهنگ پروژه دارد. شاید در پروژه‌های خیلی کوچک که خودمان تنها کاربر آن‌ها هستیم، بتوانیم بدون هماهنگی مشکل را حل کنیم. همچنین در پروژه‌ای که وضعیتش خیلی آشفته است، ممکن است تمیزکاری و اصلاحات پذیرفته شود.
3. اگر تغییر امکان‌پذیر نباشد، وضع موجود ادامه می‌یابد.

چاره‌ای جز ارائه راه‌حل‌های موقت مفید نیست. بهترین راه، معرفی راه‌حل‌های بهتر برای بی‌اثر کردن راه‌حل‌های موقتی است. امیدوارم آرامش پذیرش چیزهای غیرقابل‌تغییر، شجاعت تغییر چیزهای قابل‌تغییر و حکمت تشخیص این دو را داشته باشیم.

**رابط‌ها را به‌گونه‌ای طراحی کنید که استفاده صحیح از آن‌ها آسان و استفاده نادرست دشوار باشد**

*اسکات مایرز*

یکی از مهم‌ترین وظایف در توسعه نرم‌افزار، مشخص‌کردن رابط است. رابط‌ها در بالاترین سطح انتزاع (رابط کاربری)، پایین‌ترین سطح (رابط توابع) و سطوح میانی (رابط کلاس‌ها، کتابخانه‌ها و غیره) وجود دارند. صرف‌نظر از سطح رابط، طراحی درست آن بسیار مهم است. رابط خوب باعث افزایش بهره‌وری می‌شود درحالی‌که رابط بد منشأ خطا و نارضایتی خواهد بود. رابط باید به‌گونه‌ای طراحی شود که استفاده صحیح از آن ساده و استفاده نادرست دشوار باشد. یعنی باید به کاربر الگوها و راهنمایی‌هایی بدهیم تا راحت‌تر بتواند آن را درست استفاده کند و از اشتباهات رایج دوری کند. طراحی خوب رابط نیازمند درک عمیق نیازهای کاربر و الگوهای رایج استفاده از آن است. رابط خوب هم منجر به محصول بهتر و هم رضایت بیشتر کاربر می‌شود.

رابط‌های خوب دارای ویژگی‌های زیر هستند:

*آسان برای استفاده صحیح*

کاربران رابط خوب طراحی شده، معمولاً همیشه آن را درست استفاده می‌کنند، چون راحت‌ترین راه است. مثلاً در یک رابط گرافیکی، کاربران معمولاً روی دکمه یا منوی درست کلیک می‌کنند. در یک API خوب، معمولاً پارامترهای درست و مقادیر صحیح ارسال می‌کنند؛ چون این طبیعی‌ترین راه است. با رابط‌هایی که استفاده صحیح از آن‌ها راحت است، همه چیز به‌درستی کار می‌کند.

*دشوار برای استفاده نادرست*

رابط خوب اشتباهات احتمالی را پیش‌بینی و انجام آن‌ها را دشوار یا غیرممکن می‌کند. مثلاً یک رابط کاربری ممکن است دکمه‌های بی‌معنا در شرایط فعلی را غیرفعال کند. یا یک API ممکن است با اجازه‌دادن پاس‌دادن پارامترها به هر ترتیب، مشکل مربوط به ترتیب آرگومان‌ها را حل کند.

یک روش خوب برای طراحی رابطی که استفاده صحیح از آن آسان باشد، شبیه‌سازی آن قبل از پیاده‌سازی است. مثلاً در یک رابط کاربری می‌توان قبل از کدنویسی، آن را روی تخته سفید یا با کارت‌ها شبیه‌سازی و با آن بازی کرد. یا برای یک API می‌توان فراخوانی‌های معمول را قبل از اعلان توابع نوشت. موارد استفاده معمول را مرور کنید و مشخص کنید می‌خواهید رابط چگونه عمل کند. می‌خواهید قابلیت کلیک‌کردن روی چه چیزهایی را داشته باشید؟ می‌خواهید بتوانید چه چیزهایی را پاس دهید؟ رابطی که استفاده از آن راحت است، طبیعی به نظر می‌رسد؛ چون انجام کارهای موردنیاز را ممکن می‌کند. چنین رابطی را آسان‌تر می‌توان از دید کاربر طراحی کرد. (این دیدگاه یکی از نقاط قوت برنامه‌نویسی تست محور است.)

برای سخت کردن استفاده نادرست، باید اشتباهات احتمالی را پیش‌بینی و راه‌های جلوگیری از آن‌ها را پیدا کرد. همچنین با مشاهده استفاده‌های اشتباه، رابط را برای جلوگیری از آن‌ها اصلاح نمود. بهترین راه، غیرممکن کردن استفاده نادرست است. اگر کاربران مدام می‌خواهند یک عمل غیرقابل‌بازگشت را معکوس کنند، سعی کنید آن عمل را قابل‌بازگشت کنید. اگر مدام مقدار اشتباهی به API پاس می‌دهند، سعی کنید API را اصلاح کنید تا مقادیری را که کاربران می‌خواهند پاس دهند، بپذیرد. در نهایت باید به یاد داشت رابط‌ها برای راحتی کاربران هستند، نه پیاده‌کننده‌ها.

**آنچه نامرئی است را مرئی­تر کنید**

*جان جَگِر*

بسیاری از جنبه‌های نامرئی‌بودن به‌عنوان اصول نرم‌افزاری ستوده می‌شوند که باید حفظ شوند. اصطلاح‌شناسی ما غنی از استعارات نامرئی‌بودن است، از جمله شفافیت مکانیزم و پنهان‌سازی اطلاعات. نرم‌افزار و فرایند توسعه آن می‌تواند، بنا به گفته داگلاس آدامز، عمدتاً نامرئی باشد:

* کد منبع هیچ حضور و رفتار ذاتی ندارد و از قوانین فیزیک پیروی نمی‌کند. وقتی آن را در ادیتور بارگذاری می‌کنید مرئی می‌شود، اما اگر ادیتور را ببندید، ناپدید می‌شود. اگر بیش از حد در مورد آن فکر کنید، مانند درختی که بدون اینکه کسی بشنود می‌افتد، شروع به تردید درمورد این می‌کنید که آیا اصلاً از اول وجود داشته است یا خیر.
* یک برنامه در حال اجرا حضور و رفتار دارد، اما هیچ چیزی از سورس کدی که از آن ساخته شده را نشان نمی‌دهد. صفحه اصلی گوگل ساده و زیباست؛ اما اتفاقات پشت پرده آن قطعاً پیچیده هستند.
  + اگر %90 کار را انجام داده اید ولی بی وقفه در تلاش برای دیباگ کردن ٪10 باقی مانده هستید، پس در واقع ٪90 کارتان تمام نشده، درست است؟ رفع باگ ها باعث پیشرفت نمی شود. برای دیباگ کردن دستمزدی دریافت نمی کنید. دیباگ کردن هدر رفتن منابع است. خوب است این هدررفت را مرئی‌تر کنید تا بتوانید آن را ببینید و راه‌های جلوگیری از ایجاد آن از ابتدا را پیدا کنید.
* اگر پروژه ظاهراً در مسیر خود جلو می‌رفته؛ ولی یک هفته بعد 6 ماه عقب افتاده باشد، مشکل جدی دارید. بزرگ‌ترین مشکل این نیست که 6 ماه عقب هستید، بلکه وجود میدان‌های نیروی نامرئی قدرتمندی است که توانسته‌اند 6 ماه عقب‌ماندگی را پنهان کنند! نبود پیشرفت مرئی به معنای نبود پیشرفت واقعی است.

نامرئی‌بودن می‌تواند خطرناک باشد. وقتی چیزی ملموس برای پیونددادن تفکرتان به آن داشته باشید، واضح‌تر فکر می‌کنید. زمانی می‌توانید چیزها را بهتر مدیریت کنید که بتوانید آن‌ها را ببینید و ببینید که دائماً در حال تغییر هستند:

* + نوشتن یونیت تست، شواهدی در مورد سادگی یونیت تست کد ارائه می‌دهد. این کار به آشکارکردن وجود (یا عدم وجود) ویژگی‌های توسعه‌ای کمک می‌کند که می‌خواهید کد داشته باشد، مانند اتصال کم و انسجام بالا.
  + اجرای یونیت تست‌ها، شواهدی در مورد رفتار کد ارائه می‌دهد. این کار به آشکارکردن وجود (یا عدم وجود) ویژگی‌های اجرایی کمک می‌کند که شما می‌خواهید برنامه داشته باشد، مانند استحکام و درستی.
  + استفاده از تابلوهای اعلانات و کارت‌ها، پیشرفت را ملموس و عینی می‌کند. بدون مراجعه به ابزار مدیریت پروژه پنهان و دنبال‌کردن برنامه‌نویسان برای گزارش‌های وضعیت خیالی، وظایف می‌توانند به‌عنوان شروع‌نشده، درحال انجام یا انجام‌شده دیده شوند.
* توسعه تدریجی با افزایش شواهد توسعه فراوانی، پیشرفت (یا عدم پیشرفت) توسعه را مشهودتر می‌کند. تکمیل نرم‌افزارهای قابل‌انتشار واقعیت را نشان می‌دهد؛ برآوردها نمی‌توانند این کار را بکنند.

بهتر است نرم‌افزار با شواهد مرتب و قابل‌رؤیت فراوان توسعه پیدا کند. مشهود بودن اطمینان می‌دهد که پیشرفت واقعی است نه توهم، عمدی است نه غیرعمد، تکرارپذیر است نه تصادفی.

**ارسال پیام منجر به مقیاس‌پذیری بهتر در سیستم‌های موازی می‌شود**

*راسل ویندِر*

برنامه‌نویسان از ابتدا یاد می‌گیرند که هم‌روندی و موازی‌سازی بسیار دشوار است و حتی بهترین‌ها هم ممکن است اشتباه کنند. معمولاً تمرکز زیادی روی تِرِدها، سِمافورها، مانیتورها و دشواری دسترسی هم­روند به متغیرها، وجود دارد.

درست است که مشکلات دشوار زیادی وجود دارد و حل آن‌ها می‌تواند بسیار سخت باشد؛ اما ریشه اصلی مشکل چیست؟ حافظه مشترک. تقریباً تمام مشکلات هم‌روندی که مردم در مورد آن‌ها صحبت می‌کنند، مربوط به استفاده از حافظه مشترک قابل‌تغییر است: شرایط رقابتی، بن‌بست و غیره. پاسخ به نظر واضح است: یا از هم‌روندی صرف‌نظر کنید یا از حافظه مشترک!

احتمالاً صرف‌نظرکردن از هم‌روندی راه‌حل منطقی نیست. کامپیوترها تقریباً هر سه ماه یک‌بار هسته‌های بیشتری پیدا می‌کنند، بنابراین بهره‌برداری از موازی‌سازی واقعی روزبه‌روز مهم‌تر می‌شود. دیگر نمی‌توانیم برای بهبود عملکرد برنامه‌ها، به افزایش سرعت کلاک پردازنده‌ها تکیه کنیم. تنها راه بهبود عملکرد برنامه‌ها، استفاده از موازی‌سازی است. البته عدم بهبود عملکرد هم یک راه‌حل است، اما بعید است کاربران آن را بپذیرند.

پس آیا می‌توانیم از حافظه مشترک صرف‌نظر کنیم؟ قطعاً.

به‌جای استفاده از تردها و حافظه مشترک به‌عنوان مدل برنامه‌نویسی، می‌توانیم از پردازه‌ها و ارسال پیام استفاده کنیم. پردازه در اینجا فقط به معنای یک حالت مستقل محافظت شده با کد اجرایی است، نه لزوماً یک پردازه سیستم‌عامل. زبان‌هایی مانند Erlang (و قبل از آن occam) نشان داده‌اند که پردازه‌ها مکانیزم بسیار موفقی برای برنامه‌نویسی سیستم‌های هم‌روند و موازی هستند. چنین سیستم‌هایی تنش‌های همگام‌سازی را که سیستم‌های مولتی ترد با حافظه مشترک دارند، ندارند. علاوه بر این، مدل رسمی‌ای به نام فرایندهای متوالی ارتباطی (CSP) وجود دارد که می‌تواند به‌عنوان بخشی از مهندسی چنین سیستم‌هایی اعمال شود.

می‌توان فراتر رفت و سیستم‌های جریان داده‌ای را به‌عنوان یک روش محاسباتی معرفی کرد. در یک سیستم جریان داده‌ای، جریان کنترل به‌صورت صریح برنامه‌ریزی نشده است. به‌جای آن، یک گراف جهت‌دار از عملگرها که توسط مسیرهای داده به هم متصل شده‌اند، تنظیم می‌شود و سپس داده‌ها به سیستم تزریق می‌شوند. ارزیابی توسط آمادگی داده‌ها در سیستم کنترل می‌شود. قطعاً هیچ مشکل همگام‌سازی وجود ندارد.

بااین‌حال، زبان‌هایی مثل C، C++، جاوا، پایتون و Groovy مهم‌ترین زبان‌های توسعه سیستم‌ها هستند و به‌عنوان زبان‌هایی برای توسعه سیستم‌های مولتی ترد با حافظه مشترک به برنامه‌نویسان معرفی می‌شوند. پس چه می‌توان کرد؟ پاسخ این است که از کتابخانه‌ها و فریم‌ورک‌هایی استفاده کنیم که مدل‌های پردازه و ارسال پیام را فراهم می‌کنند و از هرگونه استفاده از حافظه مشترک قابل‌تغییر اجتناب کنیم. اگر کتابخانه و فریم‌ورکی وجود ندارد، باید ایجاد کرد.

در مجموع، برنامه‌نویسی بدون حافظه مشترک و استفاده از ارسال پیام احتمالاً موفق‌ترین روش پیاده‌سازی سیستم‌هایی است که از موازی‌سازی رایج در سخت‌افزارهای امروزی بهره می‌برند. شاید عجیب به نظر برسد، اما باوجوداینکه پردازه‌ها نسبت به تردها واحد هم‌روندی قدیمی‌تری هستند، به نظر می‌رسد آینده در استفاده از تردها برای پیاده‌سازی پردازه‌ها است.

**پیامی به آینده**

*لیندا رایزینگ*

در تمام سال‌هایی که تدریس کرده‌ و با برنامه‌نویس‌ها کار کرده‌ام، به نظر می‌رسد اکثر آن‌ها فکر می‌کنند؛ چون مسائلی که با آن‌ها درگیر هستند دشوار است، پس راه‌حل‌ها نیز باید برای همه (و حتی برای خودشان چند ماه پس از نوشتن کد) هم از نظر فهم و هم از نظر نگهداری دشوار باشد. شاید به این خاطر است که بیشتر آن‌ها افراد باهوشی هستند.

یادم هست یک‌بار جو، یکی از دانشجویانم در کلاس ساختمان داده، مجبور شد بیاید و کدی را که نوشته بود به من نشان دهد. او با شادی فریاد زد: «شرط می‌بندم نمی‌توانی حدس بزنی این کد چه کاری انجام می‌دهد!»

بدون اینکه زمان زیادی روی مثال او صرف کنم و فکر کنم چطور می‌توانم پیام مهمی را منتقل کنم، موافقت کردم. گفتم: «مطمئنم سخت روی این کار کرده‌ای. اما نکند چیز مهمی را فراموش کرده باشی؟ جو، آیا برادر کوچک‌تری نداری؟»

جو با افتخار گفت: «بله، دارم اسم او فیل است و در کلاس مقدماتی توست. او هم دارد برنامه‌نویسی یاد می‌گیرد!»

من گفتم: «عالیه. فکر می‌کنی فیل بتواند این کد را بخواند؟»

جو گفت: «اصلاً! کد سختی است!»

گفتم: «فرض کن این کد واقعی و کاربردی است و چند سال دیگر فیل استخدام شده تا آن را آپدیت کند. تو چه کمکی به او کرده‌ای؟» جو فقط به من خیره شده بود و پلک می‌زد. گفتم: «می‌دانیم فیل واقعاً باهوش است، درست است؟»

جو سر تکان داد و گفت: «درسته، من هم نسبتاً باهوش هستم!» و لبخند زد. گفتم: «اگر من نتوانم به‌راحتی آنچه را که انجام داده‌ای درک کنم و برادر باهوش‌ترت هم در درک آن مشکل داشته باشد، چه معنی می‌دهد؟» سپس پیشنهاد دادم: «چطور است هر خط کدی را که می‌نویسی، به‌عنوان پیامی برای کسی در آینده - شاید برادرت - در نظر بگیری؟ فرض کن داری به این شخص باهوش توضیح می‌دهی چطور این مسئله سخت را حل کند.»

«آیا دوست داری این‌طور تصور کنی که برنامه‌نویس باهوشی در آینده کد تو را ببیند و بگوید: «وای، عالی است؟ من کاملاً متوجه شدم اینجا چه اتفاقی افتاده و از زیبایی و ظرافت این کد شگفت‌زده شده‌ام. این کد شاهکار زیبایی است! می‌خواهم آن را به بقیه اعضای تیمم هم نشان دهم. واقعاً یک اثر هنری است!»»

«جو، فکر می‌کنی بتوانی کدی بنویسی که این مسئله سخت را حل کند و آن‌قدر زیبا باشد که مثل یک ملودی جادویی عمل کند؟ بله، درست مثل یک آواز حزن‌انگیز. فکر می‌کنم هرکسی که بتواند این راه‌حل بسیار سخت را پیدا کند، می‌تواند چیز زیبایی هم بنویسد. همم... شاید باید شروع کنم به درجه‌دهی بر اساس زیبایی؟ نظر تو چیست؟»

جو کارش را برداشت و با لبخند کوچکی به من نگاه کرد و گفت: «متوجه شدم استاد، می‌روم تا دنیا را برای فیل بهتر کنم. ممنون.»

**فرصت‌های ازدست‌رفته برای استفاده از پلی­مورفیسم**

*کرک پپردین*

پلی‌مورفیسم یکی از ایده‌های بزرگ و اساسی در برنامه‌نویسی شیءگراست. این واژه از زبان یونانی گرفته شده و به معنای اشکال (مورف) متعدد (پلی) است. در زمینه برنامه‌نویسی، پلی‌مورفیسم به اشکال مختلف یک کلاس خاص از اشیا یا متدها اشاره دارد. اما پلی‌مورفیسم صرفاً در مورد پیاده‌سازی‌های جایگزین نیست. استفاده درست از پلی‌مورفیسم، منجر به ایجاد کانتکست‌های محلی کوچک و محدودی می‌شود که بدون نیاز به بلاک‌های طولانی if-then-else می‌توانیم کار را پیش ببریم. قرارگرفتن در یک کانتکست، باعث می‌شود کار درست را مستقیماً انجام دهیم. درحالی‌که خارج از آن کانتکست مجبوریم آن را بازسازی کنیم تا بتوانیم کار درست را انجام دهیم. با استفاده دقیق از پیاده‌سازی‌های جایگزین، می‌توانیم کانتکستی ایجاد کنیم که به ما کمک کند تا کدهای کمتر و خواناتری تولید کنیم. این موضوع به بهترین وجه با چند خط کد، مانند سبد خرید ساده (غیرواقعی) زیر نشان داده می‌شود:

public class ShoppingCart {

private ArrayList<Item> cart = new ArrayList<Item>();

public void add(Item item) { cart.add(item); }

public Item takeNext() { return cart.remove(0); }

public boolean isEmpty() { return cart.isEmpty(); }

}

فرض کنید فروشگاه اینترنتی ما شامل اقلام قابل‌دانلود و اقلامی را که باید ارسال شوند. بیایید شی دیگری بسازیم که از این عملیات پشتیبانی می‌کند:

public class Shipping {

public boolean ship(Item item, SurfaceAddress address) { ... }

public boolean ship(Item item, EMailAddress address { ... }

}

هنگامی که مشتری تسویه‌حساب کرد، باید کالا را ارسال کنیم:

while (!cart.isEmpty()) {

shipping.ship(cart.takeNext(), *???*);

}

پارامتر ??? یک عملگر جدید و خاص نیست؛ سؤال این است که آیا باید آیتم را ایمیل کنم یا از طریق پست معمولی ارسال کنم. زمینه موردنیاز برای پاسخ به این سؤال دیگر وجود ندارد. ما می‌توانستیم روش ارسال را در یک متغیر boolean یا enum ذخیره کنیم و سپس با استفاده از یک بلاک if-then-else، پارامتر گمشده را پر کنیم. راه‌حل دیگر این است که دو کلاس بسازیم که هر دو از Item ارث‌بری کنند. این کلاس‌ها را DownloadableItem و SurfaceItem می‌نامیم. حالا شروع به کدنویسی می‌کنیم. من Item را به یک اینترفیس تبدیل می‌کنم که یک متد به نام ship دارد. برای ارسال محتویات سبد خرید، item.ship(shipper) را فراخوانی می‌کنیم. کلاس‌های DownloadableItem و SurfaceItem هر دو ship را پیاده‌سازی می‌کنند:

public class DownloadableItem implements Item {

public boolean ship(Shipping shipper, Customer customer) {

shipper.ship(this, customer.getEmailAddress());

}

}

public class SurfaceItem implements Item {

public boolean ship(Shipping shipper, Customer customer) {

shipper.ship(this, customer.getSurfaceAddress());

}

}

در این مثال، مسئولیت کار با Shipping را به هر آیتم واگذار کرده‌ایم. ازآنجایی‌که هر آیتم می‌داند بهترین روش ارسال آن چیست، این سازماندهی به ما این امکان را می‌دهد تا بدون نیاز به استفاده از بلاک if-then-else به کارمان ادامه دهیم. این کد همچنین استفاده از دو الگویی را نشان می‌دهد که اغلب با هم کار می‌کنند: Command و Double Dispatch. استفاده مؤثر از این الگوها به استفاده دقیق از پلی‌مورفیسم بستگی دارد. با استفاده درست از آن‌ها شاهد کاهش تعداد بلاک‌های if-then-else در کدمان خواهیم بود.

درحالی‌که در برخی موارد استفاده از if-then-else به‌جای پلی‌مورفیسم کاربردی‌تر و منطقی‌تر است، اما در اکثر موارد سبک کدنویسی پلی‌مورفیک‌تر منجر به کدبیس کوچک‌تر، خواناتر و با شکنندگی کمتر می‌شود. تعداد بلاک‌های if-then-else در کد ما نشان‌دهنده تعداد فرصت‌های ازدست‌رفته است.

**اخبار عجیب: تسترها دوستان شما هستند**

*بورک هافناگل*

چه تسترها خود را تضمین کیفیت یا کنترل کیفیت بنامند، بسیاری از برنامه‌نویسان آنها را به‌عنوان مشکل می‌شناسند. بر اساس تجربه من، برنامه‌نویس‌ها اغلب رابطه تقابلی با افرادی دارند که نرم‌افزارشان را تست می‌کنند. شکایات رایج این است که «آن‌ها خیلی سخت‌گیر هستند» و «می‌خواهند همه چیز کامل باشد». آیا این مسئله برایتان آشناست؟

من همیشه دیدگاه متفاوتی نسبت به تسترها داشته‌ام. شاید به این خاطر که در اولین شغل من، «تستر» منشی شرکت بود. مارگارت خانمی بسیار مهربان بود که دفتر را اداره می‌کرد و سعی می‌کرد به چند برنامه‌نویس جوان یاد دهد که چطور در برابر مشتریان، حرفه‌ای رفتار کنند. او همچنین استعدادی در پیداکردن هر باگی، هرچقدر هم که آن باگ پنهان بود، در لحظه داشت.

در آن زمان، روی برنامه‌ای کار می‌کردم که توسط حسابداری نوشته شده بود که فکر می‌کرد برنامه‌نویس است. نیازی به گفتن نیست که این برنامه چه مشکلات جدی داشت. وقتی فکر می‌کردم یک بخش را درست کرده‌ام، مارگارت آن را تست می‌کرد و اغلب اوقات، پس از چند کلیک و ضربه، به روش جدیدی شکست می‌خورد. گاهی اوقات ناامیدکننده و شرم‌آور بود، اما مارگارت شخصیتی خوشایند داشت و آدم خوش‌رویی بود، بنابراین هرگز به خود اجازه ندادم او را به‌خاطر بدجلوه‌دادن خودم سرزنش کنم. در نهایت، روزی فرارسید که مارگارت توانست برنامه را به طور تمیز اجرا کند، فاکتوری را وارد کند، چاپ کند و خاموشش کند. من به وجد آمدم. بهتر از آن، وقتی آن را روی سیستم مشتری نصب کردیم، همه چیز کار کرد. آن‌ها هرگز مشکلی پیدا نکردند؛ چون مارگارت قبلاً به من کمک کرده بود مشکلات را پیدا و رفع کنم.

پس این دلیلی است که می‌گویم تسترها دوستان شما هستند. ممکن است فکر کنید تسترها با گزارش مسائل جزئی، شما را بد جلوه می‌دهند. اما وقتی مشتریان از اینکه تمام آن «مسائل کوچک» را که بخش کنترل کیفیت شما را مجبور به رفع آن‌ها کرد، مزاحمشان نشد، خرسند می‌شوند، به این صورت شما عالی به نظر می‌رسید. منظور من را درک می‌کنید؟

تصور کنید این‌طوری باشد: شما در حال تست یک ابزار کاربردی هستید که از «الگوریتم‌های هوش مصنوعی پیشرو» برای پیداکردن و رفع مشکلات هم‌روندی استفاده می‌کند. شما آن را راه‌اندازی می‌کنید و بلافاصله متوجه می‌شوید که آن‌ها کلمه «intelligence» را در صفحه اولیه اشتباه تایپ کرده‌اند. کمی نامطلوب به نظر می‌رسد، اما فقط یک غلط تایپی است، درست است؟ سپس متوجه می‌شوید صفحه پیکربندی از چک‌باکس‌هایی استفاده می‌کند که باید radio button باشند، و برخی از میانبرهای صفحه‌کلید کار نمی‌کنند. حالا هیچ‌کدام از این‌ها مشکل بزرگی نیست، اما با افزایش خطاها، شما در مورد برنامه‌نویس‌ها مردد می‌شوید. اگر آن‌ها نتوانند کارهای ساده را درست انجام دهند، احتمال اینکه هوش مصنوعی که آن‌ها ساخته­اند بتواند واقعاً چیز حساسی مثل مسائل هم‌روندی را پیدا و رفع کند چقدر است؟

آن‌ها ممکن است نوابغی باشند که آن‌قدر بر روی ساختن یک هوش مصنوعی باورنکردنی متمرکز بوده‌اند که متوجه این مسائل جزئی نشده‌اند و اگر «تسترهای سخت‌گیر» به مشکلات اشاره نمی‌کردند، شما در نهایت آن‌ها را پیدا می‌کردید. حالا شما صلاحیت برنامه‌نویس‌ها را زیر سؤال برده‌اید.

بنابراین، هرچند عجیب به نظر می‌رسد، اما آن تسترهایی که مصمم‌اند حتی کوچک‌ترین باگ را در کد شما پیدا کنند، واقعاً دوستان شما هستند.

**یک فایل باینری**

*استیو فریمن*

من چند پروژه دیده‌ام که در آن‌ها فرایند ساخت، بخشی از کد را بازنویسی می‌کند تا باینری سفارشی برای هر محیط هدف تولید کند. این کار همیشه امور را پیچیده‌تر از آنچه باید باشند می‌کند و ریسک ایجاد نسخه‌های ناهمگون برای هر نصب را معرفی می‌کند. حداقل، این به معنای ساخت چندین نسخه تقریباً یکسان از نرم‌افزار است که هر کدام باید به مکان صحیح منتقل شوند و وجود اجزای متحرک بیشتر از ضرورت را نشان می‌دهد که به معنای فرصت بیشتر برای اشتباه هستند.

من یک‌بار در تیمی کار کردم که هر تغییر پراپرتی برای یک سیکل کامل ساخت باید بررسی می‌شد، بنابراین تسترها مجبور بودند هر زمان که نیاز به یک تنظیم جزئی داشتند منتظر بمانند (ذکر کنم که ساخت طولانی بود). من همچنین در تیمی کار کردم که مدیران سیستم اصرار داشتند محیط عملیاتی از ابتدا مجدداً ساخته شود (با استفاده از همان اسکریپت‌های ما) که به این معنی بود که ما هیچ شواهدی نداشتیم که نسخه در محیط عملیاتی همانی باشد که از تست‌ها عبور کرده است و غیره.

قانون ساده است: یک باینری واحد بسازید که بتوانید آن را در تمام مراحل فرایند انتشار شناسایی کنید و ارتقا دهید. جزئیات مختص محیط را در خود محیط نگه دارید. این می‌تواند به این معنی باشد که مثلاً آن‌ها را درون کانتینر اجزاء، در یک فایل شناخته شده، یا در path نگهداری کنید.

اگر تیم شما یا فرایند ساخت، کدشکن دارد یا تمام تنظیمات هدف را با کد ذخیره می‌کند، این نشان می‌دهد که هیچ‌کس طراحی را به‌اندازه کافی دقیقاً مدنظر قرار نداده است تا ویژگی‌های اصلی برنامه و ویژگی‌های خاص پلتفرم را از هم جدا کند. یا ممکن است بدتر باشد: تیم می‌داند چه باید انجام دهد اما نمی‌تواند تلاش لازم برای انجام تغییر را اولویت‌بندی کند.

البته استثناهایی وجود دارد: ممکن است شما برای اهدافی در حال ساخت باشید که به طور قابل‌توجهی محدودیت منابع متفاوتی دارند، اما این به اکثریت ما که برنامه‌های «انتقال داده از پایگاه‌داده به صفحه‌نمایش و برعکس» می‌نویسیم، مربوط نمی‌شود. یا ممکن است شما درگیر یک به‌هم‌ریختگی موروثی باشید که هم‌اکنون رفع آن خیلی سخت است. در چنین مواردی، باید به‌صورت تدریجی حرکت کنید - اما هرچه سریع‌تر شروع کنید.

و یک نکته دیگر: اطلاعات محیط را هم نسخه‌بندی کنید. چیزی بدتر از شکستن پیکربندی محیط و عدم توانایی در تشخیص اینکه چه چیزی تغییر کرده است، وجود ندارد. اطلاعات محیطی باید جدا از کد نسخه‌بندی شوند، زیرا با سرعت و دلایل متفاوتی تغییر می‌کنند. برخی تیم‌ها از سیستم‌های ورژن کنترل توزیع‌شده برای این کار استفاده می‌کنند (مانند bazaar و git)، زیرا این سیستم‌ها تغییرات انجام‌شده در محیط‌های عملیاتی - که اجتناب‌ناپذیر هستند - را به‌راحتی به ریپازیتوری بازمی‌گردانند.

**فقط کد حقیقت را می‌گوید**

*پیتر سامرلاد*

معنا و مفهوم نهایی یک برنامه توسط کد اجرایی آن مشخص می‌شود. اگر این کد تنها در فرم باینری باشد، خواندن آن دشوار خواهد بود! بااین‌حال، اگر برنامه متعلق به خودتان باشد یا یک نرم‌افزار تجاری معمولی، یک پروژه اوپن سورس، یا کدی در یک زبان تفسیر شده پویا باشد، سورس کد باید در دسترس قرار گیرد. وقتی به سورس کد نگاه می‌کنید، معنای برنامه باید آشکار باشد. برای دانستن اینکه یک برنامه چه کاری انجام می‌دهد، در نهایت تنها چیزی که می‌توانید با اطمینان به آن نگاه کنید، سورس کد است. حتی دقیق‌ترین سند الزامات هم کل حقیقت را نمی‌گوید: سند الزامات، جزئیات کامل داستان واقعی آنچه برنامه واقعاً انجام می‌دهد را شامل نمی‌شود، تنها قصد و نیت‌های سطح بالای تحلیل‌گر الزامات را دارد. یک سند طراحی ممکن است یک طراحی برنامه‌ریزی‌شده را ثبت کند، اما جزئیات لازم برای پیاده‌سازی را نخواهد داشت. این اسناد ممکن است با پیاده‌سازی فعلی همگام نباشند... یا ساده‌تر اینکه گم شده باشند. یا اصلاً نوشته نشده باشند. سورس کد ممکن است تنها چیز باقی‌مانده باشد.

با درنظرگرفتن این موضوع، از خود بپرسید که کد شما تا چه حد به شما یا هر برنامه‌نویس دیگری می‌گوید که چه کاری انجام می‌دهد.

شاید بگویید: «کامنت‌های من همه چیزی را که نیاز است بدانید به شما می‌گویند.» اما در نظر داشته باشید که کامنت‌ها کد اجرایی نیستند؛ آن‌ها می‌توانند درست مثل سایر اشکال مستندسازی اشتباه باشند. درگذشته کامنت‌ها را بدون قیدوشرط چیز خوبی می‌دانستند، بنابراین برخی برنامه‌نویسان بدون سؤال، کامنت‌های بیشتر و بیشتری می‌نویسند، حتی اگر جزئیات تکراری و واضح در کد را دوباره بیان کنند و توضیح دهند. این روش اشتباهی برای شفاف‌سازی کد است.

اگر کد شما نیاز به کامنت دارد، در مورد Refactor کردن آن به‌گونه‌ای که دیگر نیاز به کامنت نداشته باشد، فکر کنید. کامنت‌های طولانی می‌توانند فضای صفحه را شلوغ کنند و حتی ممکن است توسط IDE شما به طور خودکار پنهان شوند. اگر نیاز دارید یک تغییر را توضیح دهید، این کار را در پیام کامیت سیستم ورژن کنترل انجام دهید نه در خود کد.

چه‌کارهایی می‌توانید انجام دهید تا کد شما حقیقت را تاحدامکان واضح بیان کند؟ برای نام‌گذاری خوب تلاش کنید. کد را باتوجه‌به عملکردهای منسجم ساختاردهی کنید، این کار نام‌گذاری را نیز ساده می‌کند. کد را از هم جدا کنید تا قابلیت تعامد داشته باشد. تست‌های خودکار بنویسید تا رفتار موردنظر را توضیح دهند و اینترفیس‌ها را بررسی کنند. وقتی یاد گرفتید چطور راه‌حل ساده‌تر و بهتری را کدنویسی کنید، بدون ترحم و بی‌رحمانه Refactor کنید. کد را تا جایی که امکان دارد ساده کنید تا خواندن و درک آن آسان شود.

کد خود را مثل هر ترکیب دیگری مانند شعر، مقاله، وبلاگ عمومی یا ایمیل مهم در نظر بگیرید. آنچه را که بیان می‌کنید بادقت طراحی کنید، تا کاری را که باید انجام دهد، انجام دهد و مستقیماً آنچه را که انجام می‌دهد، نشان دهد؛ به‌گونه‌ای که وقتی دیگر حضور ندارید، هنوز هم قصد شما را منتقل کند. به یاد داشته باشید که کد مفید بسیار بیشتر از آنچه تصور می‌شد استفاده می‌شود. برنامه‌نویسانی که مسئول تعمیر و نگهداری از کد هستند از شما تشکر خواهند کرد. اگر شما یک برنامه‌نویس مسئول نگهداری هستید و کدی که روی آن کار می‌کنید به‌راحتی حقیقت را بیان نمی‌کند و خوانا نیست، دستورالعمل‌های فوق را به‌صورت پیشگیرانه اعمال کنید. مقداری نظم و سازماندهی در کد ایجاد کنید و سلامت ذهن خود را حفظ نمایید.

**مالک بودن (و Refactor کردن) فرایند بیلد**

*استیو برچوک*

این امر غیرمعمول نیست که تیم‌هایی که از نظر رویه‌های کدنویسی بسیار منضبط هستند، اسکریپت‌های بیلد را نادیده بگیرند، یا با این باور که آن‌ها صرفاً جزئیات بی‌اهمیتی هستند یا از ترس اینکه پیچیده بوده و نیاز به مراقبت توسط گروه مهندسی انتشار دارند. اسکریپت‌های بیلد غیرقابل‌نگهداری با تکرار و خطاها، مشکلاتی به اندازه کدی با فاکتورگیری ضعیف ایجاد می‌کنند.

یکی از دلایلی که توسعه‌دهندگان ماهر و منضبط، بیلد را چیزی فرعی نسبت به کار خود می‌دانند این است که اسکریپت‌های بیلد اغلب با زبانی متفاوت از سورس کد نوشته می‌شوند. دلیل دیگر این است که بیلد واقعاً «کد» نیست. این توجیهات در مقابل این واقعیت که اکثر توسعه‌دهندگان نرم‌افزار از یادگیری زبان‌های جدید لذت می‌برند، و اینکه بیلد آن چیزی است که محصولات قابل‌اجرا را برای توسعه‌دهندگان و کاربران نهایی برای تست و اجرا ایجاد می‌کند، بی‌اعتبار است. کد بدون بیلد بی‌فایده است، و بیلد آن چیزی است که معماری اجزای برنامه را تعریف می‌کند. بیلد یک بخش ضروری از فرایند توسعه است، و تصمیمات در مورد فرایند بیلد می‌تواند کد و کدنویسی را ساده‌تر کند.

اسکریپت‌های بیلد که با استفاده از اصطلاحات اشتباه نوشته شده‌اند، سخت‌تر نگهداری می‌شوند و مهم‌تر از آن، بهترشدن آن‌ها دشوار است. ارزش این را دارد که کمی وقت بگذارید تا راه درست ایجاد تغییر را درک کنید. اشکالات ممکن است هنگامی ظاهر شوند که برنامه با نسخه اشتباهی از یک وابستگی ساخته می‌شود یا زمانی که یک پیکربندی زمان بیلد اشتباه باشد.

تست‌کردن همیشه چیزی بوده است که به تیم «کنترل کیفیت» واگذار می‌شده است. اکنون متوجه شده‌ایم که تست‌کردن هنگام کدنویسی برای تحویل مقدار به‌صورت قابل‌پیش‌بینی ضروری است. به همین ترتیب، فرایند بیلد نیاز دارد که توسط تیم توسعه به مالکیت گرفته شود.

درک بیلد می‌تواند کل چرخه حیات توسعه را ساده کند و هزینه‌ها را کاهش دهد. یک فرایند ساده بیلد برای اجرا، به یک توسعه‌دهنده جدید اجازه می‌دهد سریع و آسان شروع کند. پیکربندی خودکار در بیلد می‌تواند به شما این امکان را بدهد که نتایج یکسانی در حالتی که افراد متعددی روی یک پروژه کار می‌کنند، به دست آورید و از مکالمات «برای من کار می‌کند» جلوگیری کنید. بسیاری از ابزارهای بیلد اجازه می‌دهند گزارشاتی در مورد کیفیت کد اجرا کنید که به شما اجازه می‌دهد مشکلات بالقوه را زودتر حس کنید. با صرف زمان برای درک اینکه چگونه بیلد را مال خودتان کنید، می‌توانید به خود و تمام اعضای تیمتان کمک کنید. می‌توانید روی کدنویسی ویژگی‌ها تمرکز کنید، به نفع ذی‌نفعان خود باشید و کار را لذت‌بخش‌تر کنید.

به اندازه کافی از فرایند بیلد خود بیاموزید تا بدانید چه زمانی و چگونه تغییرات را اعمال کنید. اسکریپت‌های بیلد، کد هستند. آن‌ها بسیار مهم هستند و نباید به دیگران واگذار شوند، حتی اگر دلیلی جز اینکه برنامه تا زمان بیلد کامل نیست وجود داشته باشد. کار برنامه‌نویسی تا زمانی که نرم‌افزار کاربردی تحویل ندهیم تکمیل نشده است.

**برنامه جفتی و احساس‌کردن جریان**

*گودنی هاوکنس، کاری روسلند و آن کاترین گاگنات*

تصور کنید کاملاً درگیر کاری هستید که انجام می‌دهید - متمرکز، مصمم و درگیر. احتمالاً مفهوم زمان را گم کرده‌اید. احتمالاً احساس خوشحالی می‌کنید. شما تجربه جریان (Flow) را دارید. به‌دست‌آوردن و حفظ جریان برای یک تیم کامل توسعه‌دهنده دشوار است، زیرا مزاحمت‌ها، تعاملات و حواس‌پرتی‌های زیادی وجود دارد که می‌توانند آن را به‌راحتی از بین ببرند.

اگر قبلاً برنامه‌نویسی جفتی را تمرین کرده‌اید، احتمالاً با اینکه چگونه جفت‌شدن به جریان کمک می‌کند آشنا هستید. اگر تجربه‌ای ندارید، می‌خواهیم با تجربیات خودمان شما را برای شروع ترغیب کنیم! برای موفقیت در برنامه‌نویسی جفتی، هم اعضای تیم و هم تیم به‌عنوان یک کل منسجم باید تلاش کنند.

به‌عنوان عضوی از تیم، با توسعه‌دهندگانی که از شما کم‌تجربه‌تر هستند با صبر و حوصله رفتار کنید. به ترس‌های خود درمورد اینکه توسط توسعه‌دهندگان ماهرتر تحت‌الشعاع قرار بگیرید، غلبه کنید. درک کنید افراد متفاوت هستند و این تفاوت‌ها ارزشمند است. از نقاط قوت و ضعف خودتان و سایر اعضای تیم آگاه باشید. ممکن است از آنچه همکارانتان می‌توانند به شما بیاموزند شگفت‌زده شوید.

به‌عنوان تیم، برنامه‌نویسی جفتی را برای انتشار مهارت‌ها و دانش در سراسر پروژه معرفی کنید. باید وظایف را در جفت‌ها حل کنید و مکرراً جفت‌ها و وظایف را جابه‌جا کنید. روی قاعده‌ای برای چرخش موافقت کنید. این قاعده را در صورت لزوم کنار بگذارید یا تنظیم کنید. تجربه ما این است که لزوماً نیاز نیست یک وظیفه را قبل از چرخش آن به جفت دیگر کامل کنید. قطع یک وظیفه برای انتقال آن به جفت دیگر ممکن است غیرمنطقی به نظر برسد، اما ما دریافته‌ایم که کار می‌کند.

موقعیت‌های متعددی وجود دارد که جریان کار می‌تواند مختل شود، اما برنامه‌نویسی جفتی به حفظ آن کمک می‌کند:

* «فاکتور کامیون» را کاهش می‌دهد. این یک آزمایش فکری کمی شوم است، اما اگر تعدادی از اعضای تیمتان توسط یک کامیون زیر گرفته شوند، چند نفر باقی می‌مانند که بتوانند محصول نهایی را تکمیل کنند؟ به‌عبارت‌دیگر، تحویل محصول تا چه اندازه به اعضای خاصی از تیم بستگی دارد؟ آیا دانش امتیازی است یا به اشتراک گذاشته می‌شود؟ اگر وظایف را بین جفت‌ها چرخش داده‌اید، همیشه شخص دیگری دانش لازم را دارد و می‌تواند کار را ادامه دهد. جریان تیم شما تحت‌تأثیر «فاکتور کامیون» قرار نمی‌گیرد.
* مشکلات را به طور مؤثر حل می‌کند. اگر در حال برنامه‌نویسی جفتی هستید و به مشکل سختی برمی‌خورید، همیشه کسی هست که بتوانید در مورد آن با او بحث کنید. چنین گفتگویی احتمال فراهم‌کردن امکانات بیشتری نسبت به اینکه تنها مانده باشید، دارد. هنگام چرخش کار، راه‌حل شما توسط جفت بعدی بازبینی می‌شود، پس مهم نیست اگر در ابتدا راه‌حل بهینه را انتخاب نکرده باشید.
* یکپارچه‌سازی را ساده‌تر می‌کند. اگر وظیفه فعلی شما نیازمند فراخوانی قطعه دیگری از کد است، امیدوارید نام متدها، مستندات و تست‌ها به اندازه کافی توضیحی باشند تا بتوانید درکی از کار آن به دست آورید. در غیر این صورت، جفت‌شدن با توسعه‌دهنده‌ای که در نوشتن آن کد دخیل بوده، به شما درک بهتر و یکپارچه‌سازی سریع‌تری در کد خودتان می‌دهد. علاوه بر این، می‌توانید از بحث به‌عنوان فرصتی برای بهبود نام‌گذاری، مستندات و تست استفاده کنید.
* مزاحمت‌ها را کاهش می‌دهد. اگر کسی بیاید سؤالی بپرسد، یا تلفنتان زنگ بزند، یا مجبور به پاسخ به ایمیل فوری باشید، یا باید در جلسه‌ای شرکت کنید، همکار برنامه‌نویسی جفتی‌تان می‌تواند به کدنویسی ادامه دهد. وقتی برمی‌گردید، همکارتان همچنان در جریان کار است و شما سریع می‌توانید به او ملحق شوید.
* اعضای تیم جدید را به‌سرعت در جریان کار قرار می‌دهد. با برنامه‌نویسی جفتی و چرخش مناسب جفت‌ها و وظایف، اعضای جدید تیم به‌سرعت هم با کد و هم با سایر اعضا آشنا می‌شوند.

جریان کار شما را بسیار بهره‌ور می‌کند، اما از طرفی آسیب‌پذیر هم هست. هر کاری از دستتان برمی‌آید برای به‌دست‌آوردن و حفظ آن انجام دهید!

**انواع اختصاصی دامنه را به انواع اولیه ترجیح دهید**

*اینار لاندر*

در 23 سپتامبر 1999، ماهواره 327.6 میلیون دلاری Mars Climate Orbiter در حین ورود به مدار مریخ به دلیل خطای نرم‌افزاری در زمین از دست رفت. این خطا بعداً به‌عنوان اشتباه متریک شناخته شد. نرم‌افزار ایستگاه زمینی برحسب پوند کار می‌کرد در حالی که ماهواره انتظار نیوتون داشتو این موضوع باعث شد ایستگاه زمینی قدرت پیشرانه‌های ماهواره را 4.45 برابر کمتر برآورد کند.

این مثال یکی از مثال‌های بسیار زیاد شکست نرم‌افزاری است که می‌توانست با اعمال تایپ‌های قوی‌تر و اختصاصی‌تر از آن جلوگیری شود. این مثالی از استدلال پشت بسیاری از ویژگی‌ها در زبان Ada است که یکی از اهداف اصلی طراحی آن پیاده‌سازی نرم‌افزار ایمن بحرانی جاسازی‌شده بود. Ada تایپ قوی با بررسی استاتیک برای تایپ‌های اولیه و تایپ‌های تعریف‌شده توسط کاربر دارد:

type Velocity\_In\_Knots is new Float range 0.0 .. 500.00;

type Distance\_In\_Nautical\_Miles is new Float range 0.0 .. 3000.00;

Velocity: Velocity\_In\_Knots;

Distance: Distance\_In\_Nautical\_Miles;

Some\_Number: Float;

Some\_Number:= Distance + Velocity; -- Will be caught by the compiler as a type error.

توسعه‌دهندگان در حوزه‌های کم‌تقاضاتر نیز می‌توانند از اعمال تایپ‌های دامنه­ای بیشتر سود ببرند، جایی که شاید در غیر این صورت از تایپ‌های داده‌ای اولیه ارائه شده توسط زبان و کتابخانه‌های آن مانند String و Float استفاده کنند. در جاوا، ++C‏، پایتون و سایر زبان‌های مدرن، نوع داده انتزاعی به‌عنوان کلاس شناخته می‌شود. استفاده از کلاس‌هایی مانند Velocity\_In\_Knots و Distance\_In\_Nautical\_Miles از لحاظ کیفیت ارزش زیادی به کد اضافه می‌کند:

* کد خواناتر می‌شود؛ زیرا مفاهیم یک دامنه را بیان می‌کند نه فقط Float یا String.
* کد تست‌پذیرتر می‌شود؛ زیرا رفتاری که آسان‌تر قابل تست است را کپسوله می‌کند.
* کد امکان استفاده مجدد در برنامه‌ها و سیستم‌های مختلف را فراهم می‌کند.

این رویکرد برای کاربران هر دو نوع زبان‌های تایپ ایستا و پویا به یک اندازه معتبر است. تنها تفاوت این است که توسعه‌دهندگان زبان‌های تایپ ایستا، کمکی از کامپایلر دریافت می‌کنند، درحالی‌که کسانی که زبان‌های تایپ پویا را انتخاب می‌کنند، احتمال بیشتری دارد که به یونیت تست‌های خود متکی باشند. سبک بررسی ممکن است متفاوت باشد، اما انگیزه و سبک بیان تفاوتی ندارد.

نتیجه این است که باید بررسی انواع اختصاصی دامنه را به‌منظور توسعه نرم‌افزار باکیفیت شروع کنیم.

**از خطا جلوگیری کنید**

*گیلز کولبرن*

پیغام‌های خطا مهم‌ترین تعامل بین کاربر و بقیه سیستم هستند. این پیغام‌ها زمانی رخ می‌دهند که ارتباط بین کاربر و سیستم در آستانه گسستن باشد.

راحت است که خطا را ناشی از ورودی اشتباه کاربر در نظر بگیریم. اما مردم به شیوه‌ای پیش‌بینی‌پذیر و سیستماتیک اشتباه می‌کنند؛ بنابراین می‌توان ارتباط بین کاربر و بقیه سیستم را مانند اجزای دیگر سیستم «دیباگ» کرد.

مثلاً فرض کنید کاربر باید تاریخی در محدوده مجاز وارد کند. بهتر است به‌جای اجازه‌دادن به کاربر برای واردکردن هر تاریخی، وسیله‌ای مثل لیست یا تقویم نمایش داده شود که فقط تاریخ‌های مجاز را نشان می‌دهد. این کار امکان ورود تاریخ خارج از محدوده توسط کاربر را از بین می‌برد.

خطاهای مربوط به فرمت نیز مشکل رایج دیگری هستند. به‌عنوان‌مثال اگر به کاربر یک فیلد متنی تاریخ نشان داده شود و او تاریخ قابل فهمی مثل «29 ژوئیه 2012» وارد کند، غیرمنطقی است که صرفا به این دلیل آن را رد کنیم که در فرمت ترجیحی (مثلا DD/MM/YYYY) نیست. رد کردن «29 / 07 / 2012» به دلیل داشتن فاصله اضافی بدتر است - این نوع مشکل برای کاربران به سختی قابل درک است زیرا به نظر آن ها تاریخ در فرمت مورد نظر است.

این خطا به این دلیل رخ می‌دهد که ردکردن تاریخ آسان‌تر از تجزیه‌وتحلیل سه یا چهار فرمت متداول تاریخ است. این نوع خطاهای جزئی منجر به نارضایتی کاربر و در نتیجه خطاهای بیشتری می‌شوند؛ زیرا کاربر تمرکز خود را از دست می‌دهد. به‌جای آن، باید ترجیح کاربران برای واردکردن اطلاعات، نه داده‌ها را محترم بشماریم.

راه دیگر اجتناب از خطاهای فرمت، ارائه راهنما - مثلاً برچسبی درون فیلد که فرمت موردنظر را نشان دهد («DD/MM/YYYY») - است. راهنمای دیگر می‌تواند تقسیم فیلد به سه تکست‌باکس به طول دو، دو و چهار کاراکتر باشد.

راهنماها با دستورالعمل‌ها تفاوت دارند: راهنماها معمولاً اشاره‌ای هستند؛ دستورالعمل‌ها بسیار توضیحی هستند. راهنماها در نقطه تعامل رخ می‌دهند؛ دستورالعمل‌ها قبل از نقطه تعامل ظاهر می‌شوند. راهنماها زمینه را فراهم می‌کنند؛ دستورالعمل‌ها استفاده را تعیین می‌کنند.

به‌طورکلی، دستورالعمل‌ها در جلوگیری از خطا مؤثر نیستند. کاربران معمولاً فرض می‌کنند اینترفیس‌ها مطابق با تجربیات گذشته‌شان کار می‌کنند. («مطمئناً همه می‌دانند «۲۹ ژوئیه ۲۰۱۲» به چه معناست؟»). پس دستورالعمل‌ها خوانده نمی‌شوند. اما راهنماها کاربر را از خطا دور می‌کنند.

راه دیگر اجتناب از خطا ارائه مقادیر پیش‌فرض است. مثلاً کاربران معمولاً مقادیری مطابق با امروز، فردا، تولدشان، ضرب‌الاجلشان یا تاریخی که آخرین بار در این فرم وارد کرده‌اند، وارد می‌کنند. بسته به بافت، یکی از این‌ها انتخاب خوبی به‌عنوان پیش‌فرض هوشمند است.

علت هر چه باشد، سیستم‌ها باید در برابر خطاها تحمل داشته باشند. شما می‌توانید این کار را با ارائه چند سطح بازگشت برای همه اقدامات - و به‌خصوص اقداماتی که پتانسیل ازبین‌بردن یا اصلاح داده‌های کاربر را دارند - تسهیل کنید.

ثبت و تجزیه‌وتحلیل اقدامات بازگشت همچنین می‌تواند نشان دهد کجای رابط کاربر را به سمت خطاهای ناخودآگاه سوق می‌دهد، مانند کلیک مداوم روی دکمه «اشتباه». این خطاها اغلب ناشی از راهنماهای گمراه‌کننده یا توالی تعاملاتی هستند که می‌توانید برای جلوگیری از خطاهای بیشتر آن‌ها را مجدداً طراحی کنید.

هر رویکردی را انتخاب کنید، اکثر خطاها سیستماتیک هستند - ناشی از سوءتفاهم‌ها بین کاربر و نرم‌افزار. درک نحوه تفکر، تفسیر اطلاعات، تصمیم‌گیری و ورود داده توسط کاربران به شما کمک می‌کند تا تعاملات بین نرم‌افزارتان و کاربران را دیباگ کنید.

**برنامه‌نویس حرفه‌ای**

*رابرت سی مارتین (آنکل باب)*

برنامه‌نویس حرفه‌ای کیست؟

مهم‌ترین ویژگی یک برنامه‌نویس حرفه‌ای، احساس مسئولیت شخصی است. برنامه‌نویسان حرفه‌ای مسئولیت حرفه، برآوردها، تعهدات زمان‌بندی، اشتباهات و کیفیت کار خود را برعهده می‌گیرند و آن را به دیگران واگذار نمی‌کنند.

* اگر حرفه‌ای هستید، مسئول حرفه خودتان هستید. مسئول خواندن و یادگیری هستید. مسئول همگام ماندن با صنعت و فناوری هستید. بسیاری از برنامه‌نویسان فکر می‌کنند آموزش آن‌ها وظیفه کارفرماست که کاملاً اشتباه است. آیا پزشکان و وکلا هم این‌گونه رفتار می‌کنند؟ خیر، آن‌ها در زمان شخصی و با هزینه خود آموزش می‌بینند. رابطه شما و کارفرما در قرارداد استخدام مشخص شده است: کارفرما قول دستمزد می‌دهد و شما قول انجام کار خوب.
* حرفه‌ای‌ها مسئولیت کدی را که می‌نویسند برعهده می‌گیرند. کدی را منتشر نمی‌کنند مگر اینکه مطمئن باشند درست کار می‌کند. یک دقیقه درباره‌اش فکر کنید. چطور می‌توانید خودتان را حرفه‌ای بدانید اگر حاضرید کدی را منتشر کنید که از درستی آن اطمینان ندارید؟ برنامه‌نویسان حرفه‌ای انتظار دارند بخش کیفیت مشکلی پیدا نکند چون کدشان را قبل از انتشار کاملاً تست می‌کنند. البته بخش کیفیت برخی مشکلات را پیدا خواهد کرد؛ چون هیچ‌کس کامل نیست. اما به‌عنوان یک برنامه‌نویس حرفه‌ای باید نگرشمان این باشد که هیچ‌چیز برای بخش کیفیت باقی نمی‌گذاریم.
  + حرفه‌ای‌ها بازیکنان تیم هستند. آن‌ها مسئولیت تولید کل تیم را برعهده می‌گیرند، نه فقط کار خودشان را. به یکدیگر کمک می‌کنند، به همدیگر آموزش می‌دهند، از یکدیگر یاد می‌گیرند و در صورت لزوم همدیگر را پوشش می‌دهند. وقتی یکی از اعضا ضعیف عمل کند، بقیه وارد عمل می‌شوند چون می‌دانند یک روز خودشان نیاز به پوشش خواهند داشت.
  + حرفه‌ای‌ها لیست طولانی باگ‌ را تحمل نمی‌کنند. لیست طولانی باگ کثیف است. سیستم‌هایی که هزاران مورد در پایگاه‌داده‌های ردیابی مسائل دارند فاجعه‌ای از بی‌دقتی هستند. در واقع، در بیشتر پروژه‌ها نیاز به سیستم ردیابی مسائل، نشانه‌ای از بی‌دقتی است. فقط بزرگ‌ترین سیستم‌ها باید لیست باگ به‌اندازه‌ای طولانی داشته باشند که برای مدیریت آن‌ها به اتوماسیون نیاز باشد.
  + حرفه‌ای‌ها به‌هم‌ریختگی ایجاد نمی‌کنند. آن‌ها از کیفیت کارشان احساس غرور می‌کنند. کدشان را تمیز، ساختارمند و خوانا نگه می‌دارند. از استانداردها و بهترین شیوه‌های مورد توافق پیروی می‌کنند و هرگز عجله نمی‌کنند. تصور کنید خارج از بدن خودتان هستید و می‌بینید پزشکی عمل جراحی قلب باز روی شما انجام می‌دهد. این پزشک مهلت مشخصی دارد (به معنای واقعی کلمه). او باید قبل از اینکه دستگاه بای‌پس قلبی-ریوی به تعداد زیادی از سلول های خونی شما آسیب برساند، کارش را تمام کند. شما می‌خواهید او چگونه رفتار کند؟ آیا می‌خواهید مثل یک برنامه‌نویس معمولی عمل کند، عجله کند و به هم بریزد؟ آیا می‌خواهید بگوید «این را بعداً درست می‌کنم»؟ یا می‌خواهید بادقت بر اصولش پایبند بماند، وقت بگذارد، و مطمئن باشد که رویکرد او بهترین رویکردی است که می‌تواند انجام دهد؟ آیا یک به‌هم‌ریختگی می‌خواهید یا یک کار حرفه‌ای؟

حرفه‌ای‌ها مسئولیت‌پذیر هستند. آن‌ها مسئولیت حرفه خود را برعهده می‌گیرند. مسئولیت اطمینان از کارکرد درست کدشان را برعهده می‌گیرند. مسئولیت کیفیت کارشان را برعهده می‌گیرند. هنگام نزدیک بودن مهلت انجام کار، اصول خود را کنار نمی‌گذارند. در واقع هنگام افزایش فشار، حرفه‌ای‌ها سخت‌تر از همیشه به اصولی که می‌دانند درست است پایبند می‌مانند.

**همه چیز را تحت سیستم ورژن کنترل قرار دهید**

*دیومیدیس اسپینلیس*

همه چیز را در تمام پروژه‌های خود تحت ورژن کنترل قرار دهید. ابزارهای موردنیاز در اختیار شما هستند: ابزارهای رایگان مانند Subversion، Git، Mercurial و CVS؛ فضای دیسک فراوان؛ سرورهای ارزان و قدرتمند؛ شبکه فراگیر؛ و حتی خدمات هاست پروژه. پس از نصب نرم‌افزار ورژن کنترل، تنها کاری که برای قراردادن پروژه خود در ریپازیتوری موردنیاز است، دستور مناسب در یک دایرکتوری تمیز است که حاوی کد شماست. تنها دو عملیات پایه جدید برای یادگیری وجود دارد: شما تغییرات کد خود را به ریپازیتوری کامیت می‌کنید و نسخه کاری خود را از پروژه با نسخه ریپازیتوری به‌روز می‌کنید.

وقتی پروژه شما تحت سیستم ورژن کنترل قرار گرفت، می‌توانید به طور واضح تاریخچه آن را ردیابی کنید، ببینید چه کسی چه کدی نوشته است و با یک شناسه منحصربه‌فرد به یک فایل یا نسخه پروژه ارجاع دهید. مهم‌تر از آن، می‌توانید تغییرات جسورانه کد را بدون ترس انجام دهید - دیگر نیازی به کدهای غیرفعال شده فقط برای مواردی که ممکن است در آینده نیاز داشته باشید، نیست، زیرا نسخه‌های قدیمی به طور امن در ریپازیتوری نگهداری می‌شوند. شما می‌توانید (و باید) یک نسخه نرم‌افزار را با یک نام نمادین برچسب بزنید تا بتوانید در آینده به‌راحتی نسخه دقیق نرم‌افزاری که مشتری شما اجرا می‌کند را بازبینی کنید. شما می‌توانید برنچ­های موازی توسعه ایجاد کنید: اکثر پروژه‌ها یک برنچ توسعه فعال و یک یا چند برنچ نگهداری برای نسخه‌های منتشر شده دارند که به طور فعال پشتیبانی می‌شوند.

یک سیستم ورژن کنترل اصطکاک بین توسعه‌دهندگان را به حداقل می‌رساند. زمانی که برنامه‌نویسان روی اجزای مستقل نرم‌افزار کار می‌کنند، این اجزا تقریباً به‌صورت خودکار یکپارچه می‌شوند. زمانی که برنامه‌نویسان با یکدیگر تداخل پیدا می‌کنند، سیستم متوجه می‌شود و به آنها اجازه می‌دهد تا تعارض‌ها را حل‌وفصل کنند. با اندکی تنظیمات بیشتر، سیستم می‌تواند برای هر تغییر کامیت شده، به همه توسعه‌دهندگان اطلاع‌رسانی کند و درک مشترکی از پیشرفت پروژه ایجاد نماید.

هنگام شروع و راه‌اندازی پروژه خود، خسیس نباشید: همه assetهای پروژه را تحت سیستم ورژن کنترل قرار دهید. علاوه بر سورس کد، ورژن کنترل باید مستندات، ابزارها، اسکریپت‌های ساخت، موارد تست و حتی کتابخانه‌ها را نیز شامل شود. با قرارگرفتن کامل پروژه در ریپازیتوری (که به طور مرتب پشتیبان‌گیری می‌شود)، آسیب احتمالی ناشی از ازدست‌دادن دیسک یا داده‌ها به حداقل می‌رسد. تنظیم یک ماشین جدید برای توسعه، صرفاً نیاز به بررسی‌کردن پروژه از ریپازیتوری دارد. این امر توزیع، بیلد و تست‌کردن کد در پلتفرم‌های مختلف را ساده می‌کند: در هر ماشین، یک دستور به‌روزرسانی واحد اطمینان حاصل می‌کند که نرم‌افزار نسخه فعلی است.

وقتی زیبایی کار با یک سیستم ورژن کنترل را دیدید، رعایت چند قانون باعث می‌شود شما و تیمتان حتی مؤثرتر عمل کنید:

* + هر تغییر منطقی را به‌صورت عملیات جداگانه کامیت کنید. گروه‌بندی تغییرات زیادی در یک کامیت، باعث دشوارشدن جداسازی آن‌ها در آینده می‌شود. این نکته به‌ویژه هنگام انجام بازسازی یا تغییرات سبک در سراسر پروژه مهم است چون می‌تواند سایر تغییرات را مبهم کند.
  + برای هر کامیت یک پیام توضیحی قرار دهید. حداقل، تغییرات انجام شده را به اختصار توضیح دهید، اما اگر می‌خواهید دلیل تغییر را ثبت کنید، بهترین مکان برای ذخیره آن همین‌جاست.
* در نهایت، از کامیت کردن کدی که باعث شکسته‌شدن بیلد پروژه می‌شود خودداری کنید، در غیر این صورت با سایر توسعه‌دهندگان پروژه درگیر می‌شوید.

زندگی تحت یک سیستم ورژن کنترل خیلی خوب است و نباید آن را با اشتباهات قابل‌اجتناب خراب کرد.

**ماوس را کنار بگذارید و از صفحه‌کلید فاصله بگیرید**

*بورک هافناگل*

ساعت‌ها بر روی یک مسئله پیچیده تمرکز کرده‌اید، و هیچ راه‌حلی به ذهنتان نمی‌رسد. پس بلند می‌شوید تا پاهایتان را دراز کنید یا به دستگاه‌های فروش خودکار بروید و در راه برگشت، ناگهان پاسخ به ذهنتان می‌رسد.

آیا این سناریو برایتان آشناست؟ آیا تابه‌حال فکر کرده‌اید چرا این اتفاق می‌افتد؟ رازش این است که درحالی‌که شما مشغول کدنویسی هستید، بخش منطقی مغزتان فعال است و بخش خلاق شما کنار گذاشته شده است. تا زمانی که بخش منطقی استراحت نکند، بخش خلاق نمی‌تواند چیزی را به شما ارائه دهد.

یک مثال واقعی: من مشغول تمیزکاری برخی از کدهای قدیمی بودم و با یک متد «جالب» برخورد کردم. این متد برای بررسی اینکه آیا یک استرینگ حاوی یک زمان معتبر با فرمت hh:mm:ss xx بود یا خیر، طراحی شده بود که در آن hh ساعت، mm دقیقه، ss ثانیه و xx یا AM یا PM را نشان می‌دهد.

این متد از کد زیر برای تبدیل دو کاراکتر (نمایانگر ساعت) به عدد و بررسی اینکه آیا در محدوده مجاز است یا خیر، استفاده می‌کرد:

try {

Integer.parseInt(time.substring(0, 2));

} catch (Exception x) {

return false;

}

if (Integer.parseInt(time.substring(0, 2)) > 12) {

return false;

}

همان کد دو بار دیگر نیز با تغییرات مناسب در جابه‌جایی کاراکتر و محدوده بالایی، برای آزمایش دقیقه و ثانیه ظاهر می‌شد. در انتهای متد نیز این خطوط برای بررسی AM و PM وجود داشت:

if (!time.substring(9, 11).equals("AM") &

!time.substring(9, 11).equals("PM")) {

return false;

}

اگر هیچ‌کدام از این مقایسه‌ها منجر به بازگشت مقدار false نمی‌شد، متد مقدار true را برمی‌گرداند.

اگر کد بالا به نظرتان طولانی و دشوار است نگران نباشید، به نظر من هم همین‌طور بود؛ یعنی چیزی را پیدا کرده بودم که ارزش تمیزکاری داشت. آن را Refactor کردم و چند یونیت تست نوشتم تا مطمئن شوم هنوز کار می‌کند.

وقتی تمام شد، از نتایج راضی بودم. نسخه جدید خوانا بود، از نظر اندازه نصف شده بود و دقیق‌تر هم بود؛ زیرا کد اصلی فقط محدوده بالایی برای ساعت، دقیقه و ثانیه را تست می‌کرد.

روز بعد در حال آماده‌شدن برای کار، ایده‌ای به ذهنم رسید: چرا اعتبارسنجی استرینگ را با استفاده از یک عبارت منظم انجام ندهم؟ پس از چند دقیقه تایپ‌کردن، در تنها یک خط کد پیاده‌سازی کارآمدی داشتم. این پیاده‌سازی اینجاست:

public static boolean validateTime(String time) {

return time.matches("(0[1-9]|1[0-2]):[0-5][0-9]:[0-5][0-9] ([AP]M)");

}

نکته این داستان این نیست که من در نهایت بیش از 30 خط کد را با تنها یک خط کد جایگزین کردم. نکته این است که تا زمانی که از کامپیوتر دور نشده بودم، فکر می‌کردم اولین تلاش من بهترین راه حل برای این مسئله است.

پس بار بعد که با مشکل دشواری مواجه شدید، لطفاً این لطف را در حق خودتان انجام دهید. وقتی واقعاً مسئله را درک کردید، کاری انجام دهید که بخش خلاق مغزتان را درگیر کند – مشکل را ترسیم کنید، به موسیقی گوش دهید یا فقط پیاده‌روی کنید. گاهی بهترین کاری که می‌توانید برای حل یک مسئله انجام دهید این است که ماوس را کنار بگذارید و از صفحه‌کلید دور شوید.

**کد را بخوانید**

*کاریان برگ*

ما برنامه‌نویس‌ها موجودات عجیبی هستیم. عاشق کدنویسی هستیم. اما وقتی نوبت به خواندن کد می‌رسد، معمولاً از آن دوری می‌کنیم. به‌هرحال، نوشتن کد بسیار لذت‌بخش است و خواندن کد دشوار– حتی گاهی اوقات تقریباً غیرممکن. به‌خصوص خواندن کد دیگران سخت است. نه لزوماً به این دلیل که کد دیگران بد است، بلکه به این دلیل که احتمالاً آنها به شیوه متفاوتی فکر می‌کنند و مسائل را حل می‌کنند. اما آیا تابه‌حال به این فکر کرده‌اید که خواندن کد شخص دیگری می‌تواند کدنویسی شما را بهبود بخشد؟

دفعه بعد که مشغول خواندن کدی هستید، لحظه‌ای مکث کنید و فکر کنید. آیا این کد خوانا است یا ناخوانا؟ اگر ناخوانا است، چرا؟ آیا فرمت‌بندی آن ضعیف است؟ آیا نام‌گذاری نامتناسب یا غیرمنطقی است؟ آیا چندین نگرانی مختلف در یک قطعه کد ترکیب شده‌اند؟ شاید انتخاب زبان مانع از خوانا بودن کد شده است. سعی کنید از اشتباهات دیگران درس بگیرید تا کد شما همان اشتباهات را نداشته باشد. شاید چند غافل‌گیری در انتظارتان باشد. به‌عنوان‌مثال، تکنیک‌های شکستن وابستگی ممکن است برای کاهش جفت‌شدگی خوب باشند؛ اما گاهی اوقات می‌توانند خواندن کد را سخت‌تر کنند و کدی را که برخی زیبا می‌نامند، دیگران ناخوانا می‌نامند.

اگر کد خواناست، مکث کنید ببینید آیا می‌توانید چیز مفیدی از آن یاد بگیرید یا خیر. شاید الگوی طراحی در آن استفاده شده باشد که شما از آن بی‌خبرید یا قبلاً در پیاده‌سازی آن مشکل داشته‌اید. شاید متدها کوتاه‌تر باشند و نام‌های آن‌ها گویاتر از نام‌های شما باشد. برخی از پروژه‌های اوپن­سورس پر از مثال‌های خوبی از نوشتن کدهای خوانا هستند - درحالی‌که برخی از آن ها خلاف این حرف را ثابت می کنند! کد برخی از آن‌ها را بررسی کنید و نگاهی به آن‌ها بیندازید.

خواندن کدهای قدیمی خودتان، از یک پروژه‌ای که در حال حاضر روی آن کار نمی‌کنید، می تواند تجربه خوبی باشد. از قدیمی‌ترین کدهایتان شروع کنید و به سمت کدهای جدیدتر پیش بروید. احتمالاً متوجه می‌شوید که خواندن آن‌ها مثل زمان نوشتنشان آسان نیست. کدهای اولیه شما شاید حتی خنده‌دار هم باشند، تا حدی مثل یادآوری حرف‌هایی است که شب گذشته وقتی مست بودید در میخانه زدید! نگاه کنید چگونه مهارت‌هایتان طی سال‌ها توسعه یافته‌اند - این موضوع واقعاً انگیزه‌بخش است. ببینید خواندن کدام قسمت‌ها سخت است، و فکر کنید آیا هنوز هم به همان شیوه می‌نویسید یا خیر. پس، بار بعد که احساس کردید نیاز دارید مهارت‌های برنامه‌نویسی‌تان را بهبود بخشید، کتاب دیگری نخوانید، کد بخوانید.

**علوم‌انسانی بخوانید**

*کیت بریث ویت*

تقریباً در تمام پروژه‌های توسعه به جز کوچک‌ترین آنها، مردم با مردم کار می‌کنند. در تقریباً تمام حوزه‌های تحقیقاتی به جز انتزاعی‌ترین آنها، مردم نرم‌افزاری می‌نویسند تا از مردم دیگر در رسیدن به اهدافشان پشتیبانی کنند. مردم نرم‌افزار را با مردم و برای مردم می‌نویسند. این یک کسب‌وکار مردمی است. متأسفانه، آنچه به برنامه‌نویسان آموزش داده می‌شود، اغلب آن­ها را برای کارکردن با مردمی که برای آن­ها و با آن­ها کار می‌کنند، بسیار ضعیف تجهیز می‌کند. خوشبختانه، یک حوزه کامل مطالعاتی وجود دارد که می‌تواند کمک‌کننده باشد.

به‌عنوان‌مثال، لودویگ ویتگنشتاین در کتاب *Investigations Philosophical* (وایلی بلک‌ول) و دیگر آثارش مطرح می‌کند که هر زبانی که برای صحبت‌کردن با یکدیگر استفاده می‌کنیم - نمی‌تواند - یک فرمت سریالی برای خارج‌کردن یک فکر یا ایده یا تصویر از ذهن یک نفر و انتقال آن به ذهن شخص دیگر باشد؛ بنابراین باید از همان ابتدا در مورد سوءتفاهم هنگام «گردآوری الزامات» هوشیار باشیم. ویتگنشتاین همچنین نشان می‌دهد که توانایی ما برای درک یکدیگر، نه از تعاریف مشترک، بلکه از تجربه مشترک و شکل زندگی مشترک ناشی می‌شود. این ممکن است یکی از دلایلی باشد که چرا برنامه‌نویسانی که در حوزه مسئله خود غوطه‌ور هستند، معمولاً نسبت به کسانی که از آن فاصله دارند، عملکرد بهتری دارند.

لاکُف و جانسون در کتاب *Metaphors We Live By* کاتالوگی از استعاره‌هایی که در زندگی از آن‌ها استفاده می‌کنیم را ارائه می‌دهند و پیشنهاد می‌کنند که زبان تا حد زیادی استعاری است و این استعاره‌ها بینشی را در مورد نحوه درک ما از دنیا ارائه می‌دهند. حتی اصطلاحات ظاهراً ملموسی مثل جریان نقدینگی که ممکن است در مورد یک سیستم مالی به کار ببریم، می‌تواند استعاری در نظر گرفته شود: «پول سیال است». این استعاره چگونه نحوه تفکر ما در مورد سیستم‌هایی که با پول سروکار دارند را تحت‌تأثیر قرار می‌دهد؟ یا ممکن است در مورد لایه‌هایی در یک دسته پروتکل صحبت کنیم که برخی سطح بالا و برخی سطح پایین هستند. این کاملاً استعاری است: کاربر «بالا» است و فناوری «پایین». این موضوع به تفکر ما در مورد ساختار سیستم‌هایی که می‌سازیم نفوذ می‌کند. همچنین می‌تواند نشان‌دهنده عادت تنبلی فکری باشد که گاهی اوقات شکستن آن به نفع ماست.

مارتین هایدگر به طور دقیق روش‌هایی که مردم ابزارها را تجربه می‌کنند موردمطالعه قرار داد. برنامه‌نویس‌ها ابزارها را می‌سازند و استفاده می‌کنند، در مورد آن‌ها فکر می‌کنند و آن‌ها را ایجاد، تغییر و بازآفرینی می‌کنند. ابزارها برای ما اشیای قابل‌توجهی هستند. اما همان‌طور که هایدگر در کتاب *Being and Time* نشان می‌دهد، برای کاربران، تنها زمانی یک ابزار به چیزی نامرئی تبدیل می‌شود که حین استفاده قابل‌درک است. برای کاربران، ابزارها تنها زمانی به اشیای قابل‌توجه تبدیل می‌شوند که درست کار نکنند. این تفاوت تأکید، هر زمان که قابلیت استفاده موردبحث است، ارزش درنظرگرفتن را دارد.

الینور راش مدل طبقه‌بندی ارسطویی را که با آن درک خود از دنیا را سازمان‌دهی می‌کنیم، واژگون کرد. وقتی برنامه‌نویسان از کاربران در مورد نیازمندی‌هایشان برای یک سیستم می‌پرسند، ما تمایل داریم تعاریفی مبتنی بر پیشگفتارها بخواهیم. این برای ما بسیار راحت است. اصطلاحات در پیشگفتارها به‌راحتی می‌توانند به صفاتی بر روی کلاس یا ستون‌هایی در یک جدول تبدیل شوند. این نوع طبقه‌بندی‌ها دقیق، مجزا و مرتب هستند. متأسفانه، همان‌طور که راش در *Natural Categories* و آثار بعدی نشان داد، این‌گونه نیست که مردم به‌طورکلی دنیا را درک می‌کنند. آنها دنیا را به روش‌هایی مبتنی بر مثال‌ها درک می‌کنند. برخی از مثال‌ها، به‌اصطلاح نمونه‌های اولیه، بهتر از بقیه هستند و در نتیجه طبقه‌بندی‌های حاصل مبهم هستند، دارای همپوشانی هستند و می‌توانند ساختار درونی غنی داشته باشند. تا جایی که ما بر پاسخ‌های ارسطویی اصرار داریم، نمی‌توانیم سؤالات درستی را از کاربران در مورد دنیای آن­ها بپرسیم و در دستیابی به درک مشترکی که نیاز داریم، دچار مشکل خواهیم شد.

**اغلب اوقات چرخ را دوباره اختراع کنید**

*جیسون پی سِیج*

فقط از چیزی که وجود دارد استفاده کنید - اختراع مجدد چرخ احمقانه است .…

آیا تابه‌حال این جمله یا نسخه‌های مشابه آن را شنیده‌اید؟ البته که شنیده‌اید! احتمالاً هر توسعه‌دهنده و دانشجویی به طور مکرر چنین جملاتی می‌شنود. اما چرا اختراع مجدد چرخ به‌شدت مورد انتقاد است؟ چون در اکثر موارد، کدهای موجود کدهای کارآمدی هستند. آن­ها قبلاً از نوعی کنترل کیفیت و آزمون سخت‌گیرانه عبور کرده‌اند و با موفقیت مورداستفاده قرار می‌گیرند. علاوه بر این، زمان و تلاش صرف شده برای اختراع مجدد احتمالاً مانند استفاده از یک محصول یا کدبیس موجود سودآور نیست. پس آیا باید دوباره چرخ را اختراع کنید؟ چرا؟ کِی؟

شاید نشریاتی در مورد الگوهای توسعه نرم‌افزار یا کتاب‌هایی در زمینه طراحی نرم‌افزار دیده باشید. این کتاب‌ها می‌توانند خوابیده باشند، صرف‌نظر از اینکه اطلاعات موجود در آن­ها چقدر شگفت‌انگیز است. همان‌طور که تماشای یک فیلم در مورد قایقرانی با قایقرانی بسیار متفاوت است، استفاده از کدهای موجود در مقابل طراحی نرم‌افزار خود از ابتدا، تست‌کردن، شکستن، تعمیر و بهبود آن در طول مسیر نیز بسیار متفاوت است.

اختراع مجدد چرخ صرفاً یک تمرین در مورد این نیست که سازه‌های کد را کجا قرار دهیم: بلکه در مورد کسب دانش عمیق در مورد کارکرد درونی اجزای مختلف موجود است. آیا می‌دانید مدیریت حافظه چگونه کار می‌کند؟ پیجینگ مجازی؟ آیا خودتان می‌توانید این­ها را پیاده‌سازی کنید؟ لیست‌های پیوندی مضاعف چطور؟ کلاس‌های آرایه پویا؟ کلاینت‌های ODBC؟ آیا می‌توانید یک رابط کاربری گرافیکی مشابه یک رابطی که می‌شناسید و دوست دارید بنویسید؟ آیا می‌توانید ویجت‌های مرورگر وب خودتان را بسازید؟ آیا می‌دانید چه زمانی باید یک سیستم چندگانه‌سازی‌شده در مقابل یک سیستم مولتی ترد بنویسید؟ چطور بین یک پایگاه‌داده فایلی یا حافظه‌ای تصمیم‌گیری کنید؟

اکثر توسعه‌دهندگان هرگز خودشان این نوع پیاده‌سازی‌های اصلی نرم‌افزار را انجام نداده‌اند و در نتیجه آگاهی عمیقی از نحوه کارکرد آن­ها ندارند. نتیجه این است که همه این نوع نرم‌افزارها به‌عنوان جعبه‌های سیاه اسرارآمیزی دیده می‌شوند که فقط کار می‌کنند. درک صرفاً سطح آب کافی نیست تا خطرات پنهان زیر آن را آشکار کند. ندانستن امور عمیق‌تر در توسعه نرم‌افزار توانایی شما برای ایجاد کارهای درخشان را محدود می‌کند.

اختراع مجدد چرخ و اشتباه‌کردن، ارزشمندتر از با اولین تلاش موفق شدن است. درس‌های آموخته شده از آزمون‌وخطا جنبه عاطفی دارند که خواندن یک کتاب فنی به‌تنهایی نمی‌تواند آن را ارائه دهد!

حقایق آموخته شده بسیار مهم هستند، اما تبدیل‌شدن به یک برنامه‌نویس عالی به همان اندازه که در مورد جمع‌آوری حقایق است، به کسب تجربه نیز بستگی دارد. اختراع مجدد چرخ برای آموزش و مهارت یک توسعه‌دهنده به اندازه وزنه‌برداری برای یک بدن‌ساز مهم است.

**در مقابل وسوسه الگوی Singleton مقاومت کنید**

*سام ساریست*

الگوی Singleton بسیاری از مشکلات شما را حل می‌کند. می‌دانید که تنها به یک نمونه نیاز دارید. اطمینان دارید که این نمونه قبل از استفاده مقداردهی اولیه شده است. با داشتن یک نقطه دسترسی جهانی، طراحی شما را ساده نگه می‌دارد. همه چیز خوب است. چه چیزی را در مورد این الگوی طراحی کلاسیک دوست ندارید؟

مشخص شد چیزهای خیلی زیادی! Singletonها ممکن است وسوسه‌انگیز باشند؛ اما تجربه نشان می‌دهد اکثر آن‌ها بیشتر از اینکه مفید باشند، زیان‌آور هستند. آن‌ها قابلیت تست را مختل می‌کنند و نگهداری را دشوار می‌سازند. متأسفانه این خرد جمعی اضافی آن‌طور که باید گسترده نیست و Singletonها برای بسیاری از برنامه‌نویسان همچنان غیرقابل‌مقاومت هستند. اما مقاومت در برابر آن‌ها ارزشمند است:

* نیاز به تک‌نمونه‌بودن اغلب تصوری است. در بسیاری موارد، صرفاً حدس و گمان است که در آینده نمونه‌های بیشتری نیاز نخواهد بود. پخش‌کردن چنین ویژگی‌های تصوری در سراسر طراحی برنامه باعث دردسر خواهد شد. نیازمندی‌ها تغییر می‌کنند. طراحی خوب این موضوع را می‌پذیرد اما Singletonها نمی‌پذیرند.
* Singletonها باعث ایجاد وابستگی ضمنی بین واحدهای کد مفهوماً مستقل می‌شوند. این موضوع مشکل‌آفرین است؛ چون آن‌ها پنهان هستند و جفت‌شدگی غیرضروری بین واحدها ایجاد می‌کنند. این بوی بد کد زمانی پررنگ می‌شود که سعی می‌کنید یونیت تست بنویسید که به جفت‌شدگی سست و توانایی جایگزینی انتخابی یک پیاده‌سازی جعلی به‌جای پیاده‌سازی واقعی بستگی دارد. Singletonها چنین جعلی‌سازی ساده‌ای را غیرممکن می‌کنند.
* Singletonها دارای وضعیت پایدار ضمنی هستند که دوباره یونیت تست را مختل می‌کند. یونیت تست به این بستگی دارد که تست‌ها مستقل از یکدیگر باشند، بنابراین تست‌ها می‌توانند در هر ترتیبی اجرا شوند و برنامه قبل از اجرای هر یونیت تست می‌تواند به حالت شناخته شده‌ای برگردانده شود. همین‌که Singleton با وضعیت قابل‌تغییر معرفی شدند، این کار دشوار می‌شود. علاوه بر این، چنین وضعیت پایدار قابل‌دسترسی جهانی باعث می‌شود استدلال در مورد کد، به‌ویژه در یک محیط مولتی ترد، دشوارتر شود.
* مولتی ترد بودن، گرفتاری‌های بیشتری را به الگوی Singleton اضافه می‌کند. چون قفل‌گذاری ساده روی دسترسی خیلی کارآمد نیست، الگوی موسوم به قفل‌گذاری دوبررسی (DCLP) محبوب شده است. متأسفانه این ممکن است شکل دیگری از جاذبه کشنده باشد. ثابت شده DCLP در بسیاری از زبان‌ها thread safe نیست و حتی جایی که هست همچنان فرصت‌هایی برای اشتباه مخفیانه وجود دارد.

پاک‌سازی Singletonها ممکن است چالش نهایی باشد:

* هیچ پشتیبانی برای ازبین‌بردن صریح Singletonها وجود ندارد. این موضوع می‌تواند مشکل جدی در برخی زمینه‌ها باشد - به‌عنوان‌مثال، در یک معماری پلاگین که یک پلاگین تنها زمانی به طور امن بارگیری می‌شود که تمام شیء‌های آن پاک شده باشند.
* هیچ ترتیبی برای پاک‌سازی ضمنی Singletonها در هنگام خروج از برنامه وجود ندارد. این می‌تواند برای برنامه‌هایی که حاوی Singleton با وابستگی‌های متقابل هستند، مشکل‌ساز باشد. هنگام خاموش‌کردن چنین برنامه‌هایی، یک Singleton ممکن است به Singleton دیگری دسترسی پیدا کند که قبلاً نابود شده است.

برخی از این کاستی‌ها را می‌توان با معرفی مکانیزم‌های اضافی جبران کرد. البته این کار به قیمت پیچیدگی اضافی در کدی است که می‌توانست با انتخاب یک طراحی جایگزین از آن اجتناب شود.

بنابراین، استفاده از الگوی Singleton را به کلاس‌هایی محدود کنید که واقعاً هرگز نباید بیش از یک‌بار مقداردهی اولیه شوند. از نقطه دسترسی جهانی یک Singleton در کد دلخواه استفاده نکنید. به‌جای آن، دسترسی مستقیم به Singleton باید تنها از چند مکان خوب تعریف‌شده باشد، از جایی که می‌تواند از طریق اینترفیس خود به کدهای دیگر منتقل شود. این کدهای دیگر ناآگاه هستند و در نتیجه بستگی به Singleton بودن یا نبودن پیاده‌سازی اینترفیس ندارند. این کار وابستگی‌هایی را که از یونیت تست جلوگیری می‌کرد و قابلیت نگهداری را بهبود می‌بخشد، می‌شکند. پس، بار بعد که در حال پیاده‌سازی یا دسترسی به یک Singleton هستید، امیدوارم لحظه‌ای مکث کنید و دوباره فکر کنید.

**جاده عملکرد مملو از بمب‌های کد کثیف است**

*کرک پپردین*

اغلب اوقات بهینه‌سازی عملکرد یک سیستم نیازمند تغییردادن کد است. هنگامی که نیاز به تغییر کد داریم، هر قطعه‌ای که بیش از حد پیچیده یا شدیداً جفت‌شده باشد، مانند بمب کد کثیفی است که منتظر است تا تلاش ما را منحرف کند. اولین قربانی کد کثیف، برنامه شما خواهد بود. اگر مسیر پیشرو هموار باشد، پیش‌بینی زمان اتمام کار آسان خواهد بود. مواجهه‌های غیرمنتظره با کد کثیف، ارائه پیش‌بینی منطقی را بسیار دشوار می‌کنند.

فرض کنید نقطه داغ اجرا را پیدا کرده‌اید. روند عادی، کاهش قدرت الگوریتم پایه است. فرض کنید در پاسخ به درخواست برآورد مدیر خود، پاسخ 3 تا 4 ساعت را می‌دهید. هنگام اعمال تغییر، به سرعت متوجه می‌شوید که بخش وابسته‌ای را خراب کرده‌اید. از آنجا که چیزهای تنگاتنگ مرتبط اغلب لازم است جفت شوند، احتمالاً انتظار می‌رود این خرابی رخ دهد و در برآورد حساب شده باشد. اما اگر تعمیر این وابستگی منجر به خراب شدن بخش‌های وابسته دیگر شود چه اتفاقی می افتد؟ علاوه بر این، هرچه وابستگی دورتر از مبدا باشد، احتمال کمتری وجود دارد آن را به‌عنوان چنین وابستگی بشناسید و در برآورد خود حساب کنید. ناگهان، برآورد 3 تا 4 ساعته شما می‌تواند به راحتی به 3 تا 4 هفته برسد. اغلب، این تورم غیرمنتظره در برنامه روزبه‌روز یا دو روز در میان رخ می‌دهد. این غیرمعمول نیست که ریفکتور کردن «سریع» در نهایت چندین ماه طول بکشد تا کامل شود. در این موارد، آسیب به اعتبار و سرمایه سیاسی تیم مسئول از شدید تا کشنده خواهد بود. ای کاش ابزاری برای کمک به شناسایی و اندازه‌گیری این ریسک داشتیم...

در واقع، روش‌های متعددی برای اندازه‌گیری و کنترل میزان و عمق جفت‌شدگی و پیچیدگی کد وجود دارد. از متریک‌های نرم‌افزاری می‌توان برای شمارش وقوع ویژگی‌های خاص در کد استفاده کرد. مقادیر این شمارش‌ها با کیفیت کد همبستگی دارند. دو متریک از میان چندین متریکی که جفت‌شدگی را اندازه‌گیری می‌کنند، fan-in و fan-out هستند. fan-out برای کلاس‌ها را در نظر بگیرید: این متریک به‌عنوان تعداد کلاس‌هایی تعریف می‌شود که به طور مستقیم یا غیرمستقیم از یک کلاس موردعلاقه مرجع داده شده‌اند. می‌توانید آن را به‌عنوان شمارش تمام کلاس‌هایی که باید قبل از کامپایل کلاس موردنظرتان کامپایل شوند، در نظر بگیرید. از سوی دیگر، fan-in شمارش تمام کلاس‌هایی است که به کلاس موردعلاقه وابسته هستند. با دانستن fan-out و fan-in، می‌توانیم ضریب ناپایداری را با استفاده از I = fo / (fi + fo) محاسبه کنیم. هرچه I به 0 نزدیک‌تر شود، پکیج پایدارتر می‌شود و هرچه به 1 نزدیک‌تر شود، پکیج ناپایدارتر می‌شود. پکیج‌های پایدار اهداف کم‌ریسکی برای کدنویسی مجدد هستند، درحالی‌که پکیج‌های ناپایدار احتمال بیشتری دارد حاوی بمب‌های کد کثیف باشند. هدف در ریفکتور کردن نزدیک‌تر کردن I به 0 است.

هنگام استفاده از متریک‌ها باید به یاد داشت که آن‌ها تنها قوانین تجربی هستند. صرفاً بر اساس ریاضیات، می‌بینیم که افزایش fi بدون تغییر در fo، I را به سمت 0 حرکت می‌دهد. بااین‌حال، مقدار بسیار بزرگ fan-in یک عیب دارد: این کلاس‌ها بدون خراب‌کردن وابستگی‌ها، سخت‌تر قابل‌تغییر خواهند بود. همچنین بدون رسیدگی به fan-out، واقعاً ریسک‌ها را کاهش نمی‌دهید، پس باید تعادلی اعمال شود.

یکی از عیب‌های متریک‌های نرم‌افزاری این است که آرایه عظیمی از اعداد تولیدشده توسط ابزارهای متریک می‌تواند برای افراد مبتدی ترسناک باشد. بااین‌حال، متریک‌های نرم‌افزاری می‌توانند ابزار قدرتمندی در مبارزه ما برای کد تمیز باشند. آن‌ها می‌توانند به ما کمک کنند تا بمب‌های کد کثیف را پیش از اینکه ریسک جدی برای تمرین بهینه‌سازی عملکرد باشند، شناسایی و حذف کنیم.

**سادگی از کاهش می‌آید**

*پل دبلیو هومر*

رئیسم درحالی‌که انگشتش را محکم روی کلید Delete فشار می‌داد، به من گفت: «دوباره این کار را انجام بده...». من با احساس آشنای غرق‌شدن، صفحه کامپیوتر را تماشا می‌کردم، درحالی‌که کد من - خط‌به‌خط - به‌سوی نابودی می‌رفت.

رئیسم استفان، خیلی صحبت نمی‌کرد، اما وقتی کد بد می‌دید می‌شناخت و دقیقاً می‌دانست با آن چه کند.

من به‌عنوان یک برنامه‌نویس دانشجو، با انرژی و شور فراوان اما بدون هیچ آگاهی از کدنویسی، وارد این شغل شده بودم. تمایل وحشتناکی داشتم که فکر کنم راه‌حل هر مسئله‌ای اضافه‌کردن متغیر دیگری در جایی یا افزودن خط دیگری کد است. در یک روز بد، به‌جای بهترشدن منطق با هر بازنگری، کد من تدریجاً بزرگ‌تر، پیچیده‌تر و دورتر از کار مداوم می‌شد. طبیعی است، به‌ویژه هنگامی که عجله دارید، بخواهید تغییرات بسیار کمی روی بلاک کد موجود، حتی اگر زشت باشد، اعمال کنید. اکثر برنامه‌نویس‌ها کد بد را حفظ می‌کنند، از این می‌ترسند که شروع مجدد نیازمند تلاش بسیار بیشتری نسبت به بازگشت به ابتدا باشد. این ممکن است برای کدی که نزدیک به کار است درست باشد، اما برخی کدها فراتر از هرگونه کمکی هستند.

زمان بیشتری صرف نجات کار بد می‌شود تا باید. همین که چیزی به مصرف‌کننده منابع تبدیل شد، نیاز است که سریع دور انداخته شود.

اینکه نباید به‌راحتی همه تایپ، نام‌گذاری و فرمت‌بندی را دور انداخت، درست است. اما من را مجبور کرد که در تلاش دوم (یا گاهی اوقات سوم) دوباره در مورد کد فکر کنم. بااین‌حال، بهترین رویکرد برای تصحیح کدهای بد قرارگرفتن در حالتی است که کد بی‌رحمانه ریفکتور، جابه‌جا یا حذف می‌شود.

کد باید ساده باشد. باید حداقل تعداد متغیرها، توابع، اعلان‌ها و دیگر الزامات زبانی سینتکس وجود داشته باشد. خطوط اضافی، متغیرهای اضافی... هر چیز اضافی باید فوراً حذف شود. آنچه وجود دارد، آنچه باقی مانده، باید دقیقاً به اندازه کافی برای انجام کار، کامل‌کردن الگوریتم یا انجام محاسبات باشد. هر چیز دیگری فقط نویز اضافی و ناخواسته‌ای است که به‌صورت اتفاقی معرفی شده، جریان را مخدوش می‌کند و نکات مهم را پنهان می‌کند.

البته اگر این کار را نکرد، همه را حذف و دوباره از حافظه تایپ کنید. تایپ مجدد به این شکل می‌تواند به حذف بسیاری از پیچیدگی‌های غیرضروری کمک کند.

**اصل مسئولیت واحد**

*رابرت سی مارتین (آنکل باب)*

یکی از اصول بنیادین طراحی خوب این است:

آن چیزهایی را که به یک دلیل تغییر می‌کنند، کنار هم قرار دهید و آن‌هایی را که به دلایل متفاوت تغییر می‌کنند، از هم جدا کنید.

این اصل معمولاً به‌عنوان اصل مسئولیت واحد یا SRP شناخته می‌شود. به طور خلاصه می‌گوید که یک زیرسیستم، ماژول، کلاس یا حتی تابع نباید بیش از یک دلیل برای تغییر داشته باشد. مثال کلاسیک کلاسی است که متدهایی برای منطق کسب‌وکار، گزارش‌ها و پایگاه‌داده دارد:

public class Employee {

public Money calculatePay() ...

public String reportHours() ...

public void save() ...

}

برخی برنامه‌نویسان فکر می‌کنند قراردادن این سه تابع در یک کلاس کاملاً مناسب است. ازاین‌گذشته، کلاس‌ها مجموعه‌ای از توابع هستند که بر روی متغیرهای مشترک عمل می‌کنند. اما مشکل این است که این سه تابع به دلایل کاملاً متفاوتی تغییر می‌کنند. هر زمان که قوانین تجاری برای محاسبه دستمزد انجام شود، تابع محاسبه پرداخت تغییر می‌کند. عملکرد reportHours هر زمان که شخصی فرمت متفاوتی برای گزارش بخواهد تغییر می‌کند. تابع save هر زمان که DBAها طرح پایگاه‌داده را تغییر دهند، تغییر خواهد کرد. این سه دلیل برای تغییر، کلاس Employee را بسیار ناپایدار می‌کنند. این کلاس به هرکدام از این دلایل تغییر خواهد کرد. مهم‌تر از همه، هر کلاسی که به Employee وابسته است، از این تغییرات تأثیر می‌پذیرد.

طراحی خوب سیستم به این معناست که سیستم را به مؤلفه‌هایی تفکیک کنیم که بتوانند به طور مستقل مستقر شوند. مستقرسازی مستقل به این معناست که اگر یک مؤلفه را تغییر دهیم، نیاز نیست سایر مؤلفه‌ها را دوباره مستقر کنیم. اما اگر Employee به طور گسترده توسط کلاس‌های دیگری در سایر مؤلفه‌ها استفاده شود، هر تغییری در آن احتمالاً باعث مستقرسازی مجدد سایر مؤلفه‌ها می‌شود و از مزیت اصلی طراحی مولفه‌ای (SOA) جلوگیری می‌کند. راه‌حل ساده این است که هر کلاس در مؤلفه جداگانه‌ای قرار گیرد:

public class Employee {

public Money calculatePay() ...

}

public class EmployeeReporter {

public String reportHours(Employee e) ...

}

public class EmployeeRepository {

public void save(Employee e) ...

}

هر کلاس را می‌توان در یک مؤلفه مخصوص به خود قرار داد. یا بهتر است بگوییم، تمام کلاس‌های گزارش می‌توانند وارد جزء گزارش شوند. کلاس‌های گزارش‌دهی در مؤلفه گزارش‌دهی، کلاس‌های مرتبط با پایگاه‌داده در مؤلفه ریپازیتوری و منطق کسب‌وکار در مؤلفه منطق کسب‌وکار.

خواننده تیزبین متوجه می‌شود که هنوز وابستگی‌هایی در راه‌حل بالا وجود دارد. Employee همچنان توسط سایر کلاس‌ها مورداستفاده قرار می‌گیرد. پس اگر Employee تغییر کند، احتمالاً سایر کلاس‌ها باید دوباره کامپایل و مستقر شوند؛ بنابراین Employee نمی‌تواند تغییر کند و به طور مستقل مستقر شود. اما سایر کلاس‌ها می‌توانند تغییر کنند و به طور مستقل مستقر شوند. هیچ تغییری در یکی از آن‌ها نمی‌تواند باعث کامپایل مجدد یا مستقرسازی مجدد دیگری شود. حتی Employee هم می‌تواند با استفاده دقیق از اصل وارونگی وابستگی (DIP) به‌صورت مستقل مستقر شود، اما این موضوعی برای کتاب دیگری است![[10]](#footnote-10)

استفاده دقیق از SRP، جداکردن چیزهایی که به دلایل مختلف تغییر می‌کنند، یکی از کلیدهای ایجاد طراحی‌هایی با ساختار مولفه‌ای قابل مستقرسازی مستقل است.

**از بله شروع کنید**

*الکس میلر*

مدتی پیش، در یک فروشگاه مواد غذایی به دنبال «ادامامه» (که فقط به طور مبهم می‌دانستم نوعی سبزیجات است) می‌گشتم. مطمئن نبودم که آیا این چیزی است که در بخش سبزیجات پیدا می‌کنم یا بخش مواد منجمد یا حتی در قوطی. در نهایت تسلیم شدم و یکی از کارمندان را پیدا کردم تا کمکم کند. او هم نمی‌دانست!

آن کارمند می‌توانست به شیوه‌های مختلفی واکنش نشان دهد. می‌توانست مرا به خاطر ندانستن محل موردنظر شرمنده کند، یا امکانات مبهمی بدهد، یا حتی بگوید که آن محصول را ندارند. اما به‌جای این‌ها، از درخواست من به‌عنوان فرصتی برای یافتن راه‌حل و کمک به مشتری استفاده کرد. او سایر کارکنان را صدا زد و ظرف چند دقیقه مرا به محصول دقیق در بخش مواد منجمد هدایت کرد.

آن کارمند در این مورد، درخواست را دید و از این فرض شروع کرد که ما مشکل را حل خواهیم کرد و درخواست را برآورده خواهیم کرد. او از «بله» شروع کرد نه از «نه».

وقتی اولین‌بار در نقش رهبری فنی قرار گرفتم، احساس می‌کردم وظیفه‌ام محافظت از نرم‌افزار زیبایم در برابر جریان سخیف تقاضاهایی است که از طرف مدیران محصول و تحلیلگران کسب‌وکار می‌آید. اکثر گفتگوها را با دیدن درخواست به‌عنوان چیزی که باید از آن جلوگیری کنم شروع می‌کردم نه چیزی که باید بپذیرم.

در یک لحظه، متوجه شدم شاید روش دیگری برای کار وجود دارد که صرفاً نیاز به تغییر دیدگاه من از شروع از «نه» به شروع از «بله» دارد. در واقع، به این باور رسیده‌ام که شروع از «بله» واقعاً بخشی حیاتی از یک رهبر فنی بودن است.

این تغییر ساده، رویکرد من به شغلم را به طور اساسی تغییر داد. در واقع، راه‌های زیادی برای گفتن بله وجود دارد. وقتی کسی به شما می‌گوید: «ای‌کاش این اپلیکیشن پنجره‌های گرد و شفاف داشت!» می‌توانید آن را به‌عنوان یک درخواست مضحک رد کنید. اما اغلب بهتر است از «چرا؟» شروع کنید. اغلب، دلیل واقعی و منطقی برای درخواست پنجره‌های گرد و شفاف وجود دارد. مثلاً شما در آستانه امضای قرارداد با مشتری بزرگ جدیدی هستید که پنجره‌های گرد و شفاف را اجباری کرده است.

معمولاً وقتی زمینه درخواست را می‌دانید، امکانات جدیدی باز می‌شود. اغلب می‌توان درخواست را با محصول موجود به روش دیگری برآورده کرد و بدون هیچ کاری بله گفت: «در واقع، می‌توانید پوسته پنجره‌های گرد و شفاف را دانلود و فعال کنید.»

گاهی شخص مقابل صرفاً ایده‌ای دارد که با دیدگاه شما از محصول ناسازگار است. من دریافته‌ام که معمولاً مفید است که «چرا؟» را به سمت خودتان برگردانید. گاهی بیان دلیل مشخص می‌کند واکنش اولیه شما منطقی نیست. در غیر این صورت، شاید لازم باشد موضوع را با سایر تصمیم‌گیرندگان کلیدی در میان بگذارید. به یاد داشته باشید هدف از همه این‌ها گفتن بله به شخص مقابل و تلاش برای اجرای آن است، نه فقط برای او بلکه برای شما و تیم‌تان نیز.

اگر بتوانید توضیح منطقی برای عدم سازگاری درخواست ویژگی با محصول فعلی ارائه دهید، احتمالاً گفتگوی سازنده‌ای در مورد اینکه آیا محصول درستی داریم می‌سازیم، خواهید داشت. صرف‌نظر از نتیجه این گفتگو، همه با تمرکز بیشتری روی اینکه محصول چیست و چه نیست، متمرکز خواهند شد. شروع از بله به معنای همکاری با همکاران است، نه مقابله با آنها.

**عقب‌نشینی کنید و خودکارسازی کنید، خودکارسازی کنید، خودکارسازی کنید**

*کی هورستمن*

من با برنامه‌نویسانی کار کرده‌ام که وقتی از آنها خواسته می‌شد تعداد خطوط کد یک ماژول را محاسبه کنند، فایل‌ها را در یک واژه‌پرداز کپی می‌کردند و از قابلیت «شمارش خطوط» آن استفاده می‌کردند و هفته بعد دوباره همین کار را می‌کردند. این بد بود.

من در پروژه‌ای کار کردم که فرایند پیچیده‌ای برای انتشار داشت. این فرایند شامل امضای کد و انتقال نتیجه به سرور با تعداد زیادی کلیک ماوس بود. کسی آن را خودکارسازی کرد و اسکریپت طی تست نهایی صدها بار اجرا شد، بسیار بیشتر از انتظار. این خوب بود.

پس چرا مردم وظایف مشابه را دوباره و دوباره انجام می‌دهند به‌جای اینکه عقب بکشند و زمانی را صرف خودکارسازی آن کنند؟

*اشتباه رایج 1: خودکارسازی فقط برای تست است*

البته خودکارسازی تست عالی است، اما چرا آنجا متوقف شویم؟ وظایف تکراری فراوانی در هر پروژه وجود دارد: کنترل نسخه، کامپایل، ساخت فایل JAR، تولید مستندات، انتشار، و گزارش‌دهی. برای بسیاری از این وظایف، اسکریپت نیرومندتر از ماوس است. اجرای وظایف خسته‌کننده سریع‌تر و قابل‌اعتمادتر می‌شود.

*اشتباه رایج 2: من IDE دارم، پس نیازی به خودکارسازی ندارم*

آیا تابه‌حال بحث «اما روی سیستم من کار می‌کند!» را با هم‌تیمی‌هایتان داشته‌اید؟ IDEهای مدرن هزاران تنظیم بالقوه دارند و عملاً غیرممکن است اطمینان حاصل کنید که تمام اعضای تیم پیکربندی یکسانی دارند. سیستم‌های خودکارسازی ساخت مانند Ant یا Autotools کنترل و تکرارپذیری را برای شما فراهم می‌کنند.

*اشتباه رایج 3: برای خودکارسازی باید ابزارهای عجیب و غریبی یاد بگیرم*

می‌توانید با یک زبان shell مناسب (مانند bash یا PowerShell) و یک سیستم خودکارسازی ساخت، مسیر طولانی را طی کنید. اگر نیاز به تعامل با وب‌سایت‌ها دارید، از ابزارهایی مانند iMacros یا Selenium استفاده کنید.

*اشتباه رایج 4: نمی‌توانم این وظیفه را خودکارسازی کنم چون نمی‌توانم با این فرمت‌های فایل کنار بیایم*

اگر بخشی از فرایند شما نیاز به اسناد ورد، اکسل یا تصاویر دارد، خودکارسازی آن ممکن است دشوار باشد. اما آیا این واقعاً ضروری است؟ آیا می‌توانید از متن ساده استفاده کنید؟ مقادیر جداشده با کاما؟ XML؟ ابزاری که ترسیم را از فایل متنی تولید کند؟ اغلب، کمی تغییر در فرایند می‌تواند نتایج خوبی با کاهش چشمگیر طاقت‌فرسایی به همراه داشته باشد.

*اشتباه رایج 5: وقت کافی برای فهمیدن آن را ندارم*

نیاز نیست کل bash یا Ant را یاد بگیرید تا شروع کنید. همزمان یاد بگیرید. وقتی تسکی دارید که فکر می‌کنید می‌تواند و باید خودکار شود، فقط به اندازه کافی در مورد ابزارهایتان یاد بگیرید تا آن را انجام دهید و این کار را در ابتدای پروژه انجام دهید که معمولاً پیداکردن وقت آسان‌تر است. همین که یک‌بار موفق شدید، شما (و مدیرتان) خواهید دید که سرمایه‌گذاری روی خودکارسازی منطقی است.

**از ابزارهای تحلیل کد استفاده کنید**

*سارا مانت*

ارزش تست‌کردن چیزی است که از مراحل اولیه سفر برنامه‌نویسی به توسعه‌دهندگان نرم‌افزار آموزش داده می‌شود. در سال‌های اخیر، ظهور یونیت تست، توسعه تست محور و روش‌های چابک گواهی بر علاقه فزاینده به بهره‌برداری حداکثری از تست در تمام مراحل چرخه توسعه است. بااین‌حال، تست‌کردن تنها یکی از ابزارهای بسیاری است که می‌توانید برای بهبود کیفیت کد استفاده کنید.

در گذشته‌های دور، وقتی C هنوز پدیده‌ای جدید بود، زمان CPU و ذخیره‌سازی هر نوعی بسیار باارزش بودند. اولین کامپایلرهای C به این موضوع توجه داشتند و تعداد عبور از کدی که نوشته بودند را با حذف برخی تجزیه‌وتحلیل‌های معنایی کاهش داده بودند. این به معنای آن بود که کامپایلر تنها زیرمجموعه کوچکی از باگ‌هایی را که می‌توانستند در زمان کامپایل تشخیص داده شوند، بررسی می‌کرد. برای جبران این موضوع، استفن جانسون ابزاری به نام lint نوشت که پرزهای کد شما را تمیز می‌کند - که برخی از تجزیه‌وتحلیل‌های ایستا که از کامپایلر C خواهر آن حذف شده بود را پیاده‌سازی کرد. بااین‌حال، ابزارهای تجزیه‌وتحلیل ایستا شهرتی برای ارائه هشدارهای مثبت کاذب بسیار زیاد و هشدارهایی در مورد قراردادهای سبکی که لزوماً نیاز نیست پیروی شوند، به دست آوردند.

چشم‌انداز فعلی زبان‌ها، کامپایلرها و ابزارهای تجزیه‌وتحلیل ایستا بسیار متفاوت است. حافظه و زمان CPU اکنون نسبتاً ارزان هستند، بنابراین کامپایلرها می‌توانند خطاهای بیشتری را بررسی کنند. تقریباً هر زبانی حداقل یک ابزار دارد که نقض راهنماهای سبک، مشکلات رایج و گاهی اوقات خطاهای حیله‌گرانه‌ای را که می‌توانند به‌سختی شناسایی شوند، مانند ارجاعات احتمالی null اشاره‌گر را بررسی می‌کند.

ابزارهای پیشرفته‌تر مانند Splint برای C یا Pylint برای Python قابل پیکربندی هستند، به این معنی که می‌توانید انتخاب کنید کدام خطاها و هشدارها توسط ابزار با یک فایل پیکربندی، از طریق سوئیچ‌های خط فرمان یا در IDE خود نشان داده شوند. Splint حتی به شما اجازه می‌دهد کد خود را در توضیحات برای دادن راهنمایی‌های بهتر در مورد کارکرد برنامه شما توضیح دهید.

اگر همه چیز دیگر شکست خورد، و شما خودتان را در حال جستجو برای اشکالات ساده یا نقض استانداردهایی می‌بینید که توسط کامپایلر، IDE، یا ابزارهای lint شما شناسایی نشده‌اند، پس همیشه می‌توانید خودتان یک بررسی‌کننده­ی ایستا درست کنید. این کار به آن سختی که به نظر می‌رسد، نیست. اکثر زبان‌ها، به‌ویژه آن‌هایی که به‌عنوان زبان‌های پویا شناخته می‌شوند، درخت معناشناسی انتزاعی و ابزارهای کامپایلر خود را به‌عنوان بخشی از کتابخانه استاندارد خود ارائه می‌دهند. آشناشدن با گوشه و کنارهای گرد و غبارگرفته­ی کتابخانه‌های استانداردی که توسط تیم توسعه­ی زبانی که استفاده می‌کنید، استفاده می‌شود، ارزش بسیار زیادی دارد، زیرا این‌ها اغلب حاوی جواهرهای پنهانی هستند که برای تجزیه‌وتحلیل ایستا و آزمون پویا مفیدند. به‌عنوان‌مثال، کتابخانه استاندارد پایتون حاوی یک disassembler است که به شما می‌گوید بایت‌کد مورداستفاده برای تولید برخی از کدهای کامپایل شده یا شیء کد چیست. این به نظر می‌رسد یک ابزار مبهم برای نویسندگان کامپایلر در تیم python-dev باشد، اما در واقع به طور غیرمنتظره‌ای در موقعیت‌های روزمره مفید است. یک چیزی که این کتابخانه می‌تواند disassemble کند، آخرین ردیابی پشته شماست که بازخوردی در مورد دقیقاً کدام دستورالعمل بایت‌کد آخرین استثنای ربوده نشده را انداخته است.

بنابراین، اجازه ندهید که آزمایش پایان تضمین کیفیت شما باشد - از ابزارهای تجزیه‌وتحلیل استفاده کنید، و از ساختن ابزار خودتان نترسید.

**تست رفتار موردنیاز، نه رفتار اتفاقی**

*کِوین هِنی*

یک خطای رایج در نوشتن تست‌ها این است که فرض کنیم آنچه یک پیاده‌سازی انجام می‌دهد دقیقاً همان چیزی است که باید تست شود. در نگاه اول این مسئله به نظر یک فضیلت می‌رسد نه یک دام. اما وقتی به شکل دیگری بیان شود مشکل واضح‌تر می‌شود: یک اشتباه رایج در تست‌نویسی اتصال ثابت تست‌ها به جزئیات پیاده‌سازی است درحالی‌که این جزئیات اتفاقی هستند و هیچ ارتباطی با عملکرد موردنظر ندارند.

وقتی تست‌ها به جزئیات پیاده‌سازی وابسته باشند، تغییراتی در پیاده‌سازی که در واقع با رفتار موردنیاز سازگارند ممکن است منجر به شکست تست‌ها و بروز مثبت کاذب شود. برنامه‌نویسان معمولاً یا تست‌ها را دوباره می‌نویسند یا کد را، با این فرض اشتباه که مثبت کاذب واقعی است. این کار باعث می‌شود رفتارهای اتفاقی به رفتارهای موردنیاز تبدیل شوند. در بازنویسی یک تست، برنامه‌نویسان یا تست را دوباره روی رفتار موردنیاز متمرکز می‌کنند (خوب است) و یا به‌سادگی آن را به پیاده‌سازی جدید متصل می‌کنند (خوب نیست). تست‌ها باید هم دقیق و هم دارای دقت کافی باشند.

به‌عنوان‌مثال، در یک مقایسه سه‌جانبه مانند String.compareTo در جاوا یا strcmp در زبان C، الزام این است که اگر سمت چپ کوچک‌تر باشد منفی باشد، اگر بزرگ‌تر باشد مثبت باشد و اگر برابر باشند صفر باشد. این سبک مقایسه در بسیاری از APIها از جمله تابع مقایسه‌گر qsort در C و compareTo در اینترفیس Comparable در جاوا استفاده می‌شود. اگرچه مقادیر خاص -1 و +1 اغلب به ترتیب برای نشان دادن کوچکتر و بزرگ‌تر استفاده می‌شوند، اما برنامه‌نویسان اشتباهاً فرض می‌کنند این‌ها الزامات واقعی هستند و در نتیجه تست‌هایی می‌نویسند که این فرض را ثابت می‌کنند.

مسئله مشابهی در تست‌هایی که به وجود می‌آید فاصله‌گذاری، دقیقاً کلمات انتخاب شده و سایر جنبه‌های قالب‌بندی و ارائه متنی را که اتفاقی هستند، مورد ادعا قرار می‌دهند. مگر اینکه شما برای مثال یک تولیدکننده XML هستید که قالب‌بندی قابل پیکربندی ارائه می‌دهد، فاصله‌گذاری نباید در نتیجه تأثیر چندانی داشته باشد. همچنین متصل‌کردن ثابت مکان دکمه‌ها و برچسب‌ها روی کنترل‌های UI، گزینه تغییر و بهبود این جزئیات اتفاقی در آینده را کاهش می‌دهد. تغییرات جزئی در پیاده‌سازی و تغییرات کم‌اهمیت در قالب‌بندی ناگهان به شکست در بیلد تبدیل می‌شوند.

تست‌های بیش از حد مشخص‌شده اغلب مشکلی با رویکردهای وایت باکس به تست واحد هستند. تست‌های وایت باکس از ساختار کد برای تعیین موارد تست موردنیاز استفاده می‌کنند. حالت شکست معمول تست وایت باکس این است که تست‌ها در نهایت ادعا می‌کنند کد آنچه را که انجام می‌دهد، انجام می‌دهد. صرفاً بیان مجدد آنچه از کد کاملاً مشهود است، هیچ ارزش‌افزوده‌ای ندارد و منجر به ایجاد حس کاذبی از پیشرفت و امنیت می‌شود.

برای مؤثربودن، تست‌ها نیاز دارند تا الزامات قراردادی را بیان کنند نه صرفاً پیاده‌سازی‌ها را تقلید کنند. آن‌ها نیاز دارند تا دید بلک­باکسی نسبت به واحدهای تحت تست داشته باشند و قراردادهای رابط را به شکل اجرایی طراحی کنند؛ بنابراین، رفتار تست شده را با رفتار موردنیاز هم‌راستا کنید.

**دقیق و ملموس تست کنید**

*کوین هنی*

تست‌کردن رفتار موردنیاز و ضروری یک بخش کد به‌جای رفتار تصادفی پیاده‌سازی خاص آن، حائز اهمیت است. اما این نباید بهانه‌ای برای نوشتن تست‌های مبهم باشد. تست‌ها باید هم دقیق و هم صحیح باشند.

الگوریتم‌های مرتب‌سازی که چیزی شبیه یک کلاسیک آزموده شده و تست شده هستند، مثال توضیحی خوبی ارائه می‌دهند. پیاده‌سازی آن‌ها شاید تکلیف روزمره‌ای برای برنامه‌نویس نباشد، اما مفهوم مرتب‌سازی برای اکثریت آشناست. بااین‌حال، این آشنایی ممکن است دیدن فراتر از برخی فرضیات را دشوار کند.

وقتی از برنامه‌نویس‌ها سؤال می‌شود «چه چیزی را تست می‌کنید؟»، پاسخ رایج «نتیجه مرتب شده» است. درحالی‌که درست است، اما کامل نیست. وقتی برای یک شرط دقیق‌تر سؤال می‌شود، بسیاری از برنامه‌نویسان اضافه می‌کنند که دنباله نتیجه باید همان طول دنباله اصلی باشد. اگرچه درست است، اما هنوز هم کافی نیست. به‌عنوان‌مثال، باتوجه‌به دنباله زیر:

3 1 4 1 5 9

دنباله زیر نیز پس از مرتب‌سازی غیرنزولی، شرط طول برابر و مرتب بودن را برآورده می‌کند:

3 3 3 3 3 3

اگرچه این مشخصه‌ها را برآورده می‌کند، اما قطعاً آن چیزی نیست که در نظر گرفته شده است! این مثال بر اساس خطایی از کد تولیدی واقعی گرفته شده (خوشبختانه قبل از انتشار آن شناسایی شد) که در آن یک لغزش ساده در تایپ یا لحظه‌ای عدم تمرکز منطقی منجر به یک مکانیزم پیچیده برای پر کردن کل نتیجه با اولین عنصر آرایه داده‌شده شده بود. شرط کامل پس از اجرا این است که نتیجه مرتب شده باشد و یک جایگشت از مقادیر اصلی را نگه دارد. این به شکل مناسبی رفتار موردنیاز را محدود می‌کند. اینکه طول نتیجه برابر با طول ورودی است در کل فرایند حاصل می‌شود و نیازی به بیان مجدد آن نیست.

برای داشتن یک تست خوب، حتی بیان شرط پس از اجرا به روش توضیح داده شده کافی نیست. یک تست خوب باید خوانا باشد. باید به اندازه کافی قابل‌فهم و ساده باشد که بتوانید به‌راحتی ببینید درست است یا نه. مگر اینکه از قبل کدی برای بررسی داشته باشید که نشان بدهد یک دنباله مرتب شده و اینکه یک دنباله حاوی جایگشتی از مقادیر دنباله دیگر است یا نه. بسیار احتمال دارد که کد تست پیچیده‌تر از کد تحت تست باشد. همان‌طور که تونی هوآر گفت:

دو راه برای ساختن طراحی نرم‌افزار وجود دارد: یک راه این است که آن را آن‌قدر ساده کنید که قطعاً هیچ نقصی نداشته باشد و راه دیگر این است که آن را آن‌قدر پیچیده کنید که هیچ نقص مشهودی نداشته باشد.

استفاده از مثال‌های عینی این پیچیدگی تصادفی و فرصت برای بروز مشکل را از بین می‌برد. به‌عنوان‌مثال، باتوجه‌به دنباله زیر:

3 1 4 1 5 9

نتیجه مرتب‌سازی به شرح زیر است:

1 1 3 4 5 9

هیچ پاسخ دیگری قابل‌قبول نیست. هیچ جایگزینی را نپذیرید.

مثال‌های عینی به طور قابل‌دسترس و بدون ابهام، به توضیح رفتار عمومی کمک می‌کنند. نتیجه اضافه‌کردن یک آیتم به یک مجموعه خالی صرفاً این نیست که دیگر خالی نیست: بلکه این است که مجموعه اکنون یک آیتم دارد و آن آیتم واحد، همان آیتم اضافه شده است. دو یا چند آیتم ممکن است به‌عنوان غیر خالی در نظر گرفته شوند، اما اشتباه خواهند بود. یک آیتم واحد با مقدار متفاوت نیز اشتباه خواهد بود. نتیجه افزودن یک ردیف به یک جدول صرفاً این نیست که جدول یک ردیف بزرگ‌تر است؛ بلکه این است که می‌توان از کلید آن ردیف برای بازیابی ردیف اضافه شده استفاده کرد و... .

در تعیین رفتار، تست‌ها نه‌تنها باید دقیق باشند: بلکه باید صحیح نیز باشند.

**تست هنگام خواب (و آخر هفته‌ها)**

*راجیت آتاپاتو*

آرام باشید. منظور من مراکز توسعه دور از دسترس، اضافه‌کاری در آخر هفته‌ها یا کار در شیفت شب نیست. بلکه می‌خواهم توجه شما را به میزان قدرت محاسباتی که در اختیار داریم، جلب کنم. به طور خاص، اینکه چقدر از آن برای آسان‌تر کردن زندگی خود به‌عنوان برنامه‌نویس استفاده نمی‌کنیم. آیا شما به طور مداوم در یافتن قدرت محاسباتی کافی در طول روز کاری با مشکل مواجه هستید؟ اگر چنین است، سرورهای تست شما در خارج از ساعات کاری عادی چه کاری انجام می‌دهند؟ در اکثر مواقع، سرورهای تست در طول شب و آخر هفته بیکار هستند. شما می‌توانید از این موضوع به نفع خود استفاده کنید.

* آیا تابه‌حال مرتکب ارسال یک تغییر بدون اجرای همه تست‌ها شده‌اید؟ یکی از دلایل اصلی عدم اجرای سوئیت تست‌ها قبل از ارسال کد توسط برنامه‌نویسان، طولانی‌بودن زمان اجرای آن‌هاست. وقتی مهلت‌ تحویل نزدیک می‌شود و فشار افزایش می‌یابد، انسان‌ها به طور طبیعی شروع به بریدن گوشه‌ها می‌کنند. یک راه برای رسیدگی به این مسئله، تقسیم سوئیت تست بزرگ به دو یا چند پروفایل است. یک پروفایل تست اجباری کوچک‌تر که سریعاً اجرا می‌شود به اطمینان از اجرای تست‌ها قبل از هر کامیت کمک می‌کند. همه پروفایل‌های تست (از جمله پروفایل اجباری - فقط برای اطمینان) می‌توانند به‌صورت خودکار در طول شب اجرا شوند تا نتایج خود را صبح گزارش دهند.
* آیا فرصت کافی برای تست پایداری محصول خود داشته‌اید؟ اجرای تست‌های طولانی‌مدت برای شناسایی نشت حافظه و سایر مشکلات پایداری، حیاتی است. این تست‌ها به‌ندرت در طول روز اجرا می‌شوند، زیرا زمان و منابع را محدود می‌کنند. شما می‌توانید یک تست نفوذ را برای اجرا در طول شب و مدت طولانی‌تری در طول آخر هفته اتوماتیک کنید. از ساعت 6 بعدازظهر جمعه تا ساعت 6 صبح روز دوشنبه بعد، 60 ساعت زمان تست بالقوه وجود دارد.
* آیا زمان کیفی کافی برای محیط تست عملکرد خود دارید؟ من دیده‌ام که تیم‌ها برای گرفتن زمان روی محیط تست عملکرد با یکدیگر مشاجره می‌کنند. در اکثر موارد، هیچ‌یک از تیم‌ها در طول روز زمان کافی و باکیفیت ندارند، درحالی‌که این محیط بعد از ساعات اداری عملاً بیکار است. سرورها و شبکه در طول شب یا آخر هفته آن‌قدر شلوغ نیستند، پس زمان ایده‌آلی برای اجرای برخی تست‌های عملکرد باکیفیت است.
* در بسیاری از موارد، محصول شما هدف اجرا بر روی انواع پلتفرم‌ها را دارد. به‌عنوان‌مثال، هم 32 بیتی و هم 64 بیتی، روی لینوکس، سولاریس و ویندوز، یا صرفاً روی نسخه‌های مختلف یک سیستم‌عامل. بدتر از آن، بسیاری از برنامه‌های مدرن خود را در معرض انبوهی از مکانیزم‌ها و پروتکل‌های انتقال (HTTP، AMQP، SOAP، CORBA و غیره) قرار می‌دهند. تست دستی تمام این ترکیبات بسیار وقت‌گیر است و احتمالاً به دلیل فشار منابع، نزدیک به یک انتشار انجام می‌شود. متاسفانه، ممکن است در چرخه برای گرفتن برخی از باگ های بدجنس خیلی دیر باشد.

اجرای خودکار تست‌ها در طول شب یا آخر هفته‌ها این اطمینان را می‌دهد که تمام این جایگشت‌ها بیشتر تست شوند. با کمی فکرکردن و دانش اسکریپت‌نویسی، می‌توانید چند کار cron را برنامه‌ریزی کنید تا تست‌ها را در طول شب و آخر هفته اجرا کنند. ابزارهای تست فراوانی نیز وجود دارد که می‌توانند کمک کنند. برخی از سازمان‌ها حتی دارای گریدهای سرور هستند که سرورها را در بخش‌ها و تیم‌های مختلف جمع می‌کنند تا از استفاده بهینه از منابع مطمئن شوند. اگر این مورد در سازمان شما موجود باشد، می‌توانید تست‌ها را برای اجرا در شب یا آخر هفته ارسال کنید.

**تست‌کردن، دقت مهندسی توسعه نرم‌افزار است**

*نیل فورد*

برنامه‌نویسان دوست دارند هنگام توضیح کار خود به غیرفنی‌ها، از استعاره‌هایی مانند ساختن پل استفاده کنند تا به نظر برسد کارشان مهندسی «سخت» است. اما این استعاره‌ها زمانی که بخواهیم آن‌ها را خیلی جدی بگیریم، شکست می‌خورند - توسعه نرم‌افزار در بسیاری جهات مهم با مهندسی «سخت» متفاوت است.

در مقایسه با مهندسی «سخت»، دنیای توسعه نرم‌افزار تقریباً در همان جایی قرار دارد که سازندگان پل زمانی آنجا بودند که استراتژی متداول ساختن یک پل و سپس غلتاندن چیزی سنگین روی آن بود. اگر پل پابرجا می‌ماند، پل خوبی بود؛ در غیر این صورت، خب، نیاز به طراحی دوباره بود. طی چند هزار سال گذشته، مهندسان ابزارهای ریاضی و فیزیکی را توسعه داده‌اند که می‌توانند بدون نیاز به ساخت آن برای دیدن اینکه چه کاری انجام می‌دهد، برای یک راه‌حل سازه‌ای استفاده کنند. ما چنین چیزی در نرم‌افزار نداریم و احتمالاً هرگز نخواهیم داشت. علت این تفاوت اساسی در ماهیت نرم‌افزار است. برای یک بررسی عمیق از مقایسه مهندسی نرم‌افزار و مهندسی معمولی، «چیستی طراحی نرم‌افزار» نوشته شده توسط جک ریوِز در *ژورنال C ++* در سال 1992 یک کلاسیک است.[[11]](#footnote-11)\* اگرچه تقریباً دو دهه پیش نوشته شده است، اما هنوز هم بسیار دقیق است. ریوز تصویری تیره‌وتار از این مقایسه ارائه کرد، اما چیزی که در سال 1992 از دست رفته بود، اخلاقیات قوی تست برای نرم‌افزار بود.

تست‌کردن چیزهای «سخت» دشوار است؛ زیرا لازم است آن­ها را بسازیم تا بتوانیم تست کنیم، درحالی‌که ساختن صرفاً برای تست‌کردن امکان‌پذیر نیست. اما در نرم‌افزار، فرایند ساخت بسیار کم‌هزینه است. ما ابزارهایی برای تسهیل این کار داریم؛ مانند یونیت تست، شیء‌های ساختگی زیرساخت‌های تست و بسیاری موارد دیگر. سایر مهندسان دوست دارند بتوانند چیزی بسازند و در شرایط واقع‌گرایانه تست کنند، اما به‌عنوان توسعه‌دهندگان نرم‌افزار باید تست‌کردن را به‌عنوان مکانیزم اصلی (اگرچه نه‌تنها مکانیزم) تأیید کیفیت بپذیریم. به‌جای انتظارکشیدن برای نوعی محاسبات برای نرم‌افزار، اکنون ابزارهای لازم را در اختیار داریم تا از عملیات مهندسی خوب اطمینان حاصل کنیم. از این منظر، اکنون مهماتی در برابر مدیرانی داریم که به ما می‌گویند «وقت تست‌کردن نداریم». هرگز یک سازنده پل از مدیر خود نمی‌شنود: «زحمت تحلیل سازه‌ای آن ساختمان را نکشید- مهلت کمی داریم.» شناخت اینکه تست‌کردن در واقع مسیر تکرارپذیری و کیفیت در نرم‌افزار است، عنوان توسعه‌دهندگان اجازه می‌دهد در برابر استدلال علیه آن به‌عنوان غیرحرفه‌ای مسئولیت‌پذیر واکنش نشان دهیم.

باتوجه‌به ابزارهای موجود، دیگر نمی‌توانیم تست‌کردن را بهانه کنیم. تست‌کردن همانند تجزیه‌وتحلیل سازه‌ای، زمان‌بر است؛ اما کیفیت محصول نهایی را تضمین می‌کند. تست‌کردن بخش جدایی‌ناپذیر توسعه مسئولانه نرم‌افزار است.

**فکرکردن به استیت‌ها**

*نیکلاس نیلسون*

مردم در دنیای واقعی رابطه عجیبی با استیت دارند.

امروز صبح، به فروشگاه محلی سر زدم تا برای یک روز دیگر از تبدیل کافئین به کد آماده شوم. ازآنجایی‌که روش موردعلاقه من برای انجام این کار نوشیدن لاته است و نتوانستم هیچ شیری پیدا کنم، از کارمند پرسیدم.

او گفت: «متأسفم، ما فوق‌العاده، خیلی‌خیلی شیر نداریم.»

برای یک برنامه‌نویس، این یک بیان عجیب است. یا شیر دارید یا ندارید. هیچ مقیاسی وجود ندارد که نشان دهد چقدر شیر ندارید. شاید او سعی داشت بگوید که آن‌ها برای یک هفته شیر نخواهند داشت، اما نتیجه یکسان بود – روز اسپرسو برای من.

در بیشتر موقعیت‌های دنیای واقعی، نگرش آسان‌گیرانه افراد نسبت به استیت مشکلی ایجاد نمی‌کند. اما متأسفانه، بسیاری از برنامه‌نویسان نیز در مورد استیت‌ها بسیار مبهم هستند – و این یک مشکل است.

یک فروشگاه اینترنتی ساده را در نظر بگیرید که فقط کارت‌های اعتباری را می‌پذیرد و به مشتریان فاکتور نمی‌دهد، با یک کلاس Order حاوی این متد:

public boolean isComplete() {

return isPaid() && hasShipped();

}

منطقی به نظر می‌رسد، درست است؟ خب، حتی اگر عبارت به‌جای کپی و چسباندن همه‌جا، به طور خوبی در یک متد اکسترکت شده باشد، این عبارت نباید اصلاً وجود داشته باشد. واقعیت این است که یک سفارش نمی‌تواند قبل از پرداخت آن ارسال شود؛ بنابراین، hasShipped نمی‌تواند درست باشد مگر اینکه isPaid درست باشد که بخشی از عبارت را اضافی می‌کند. هنوز ممکن است بخواهید isComplete را برای وضوح در کد داشته باشید، اما در این صورت باید به این شکل باشد:

public boolean isComplete() {

return hasShipped();

}

در کارم همیشه هم بررسی‌های ازدست‌رفته و هم بررسی‌های اضافی را می‌بینم. این مثال بسیار کوچک است، اما وقتی لغو و بازپرداخت را اضافه کنید، پیچیده‌تر می‌شود، و نیاز به مدیریت خوب استیت افزایش می‌یابد. در این مورد، یک سفارش فقط می‌تواند در یکی از سه استیت مجزا باشد:

* در حال انجام: می‌توانید اقلام اضافه یا حذف کنید. نمی‌توان ارسال کرد.
* پرداخت شده: نمی‌توانید اقلام اضافه یا حذف کنید. می‌تواند ارسال شود.
* ارسال شده: انجام شده است. هیچ تغییر دیگری پذیرفته نمی‌شود.

این استیت‌ها مهم هستند، و شما نیاز دارید قبل از انجام عملیات‌ها، بررسی کنید که در استیت مورد انتظار هستید، و فقط از استیت فعلی به یک استیت قانونی تغییر وضعیت دهید. به طور خلاصه، باید از اشیا خود بادقت، در مکان‌های درست، محافظت کنید.

اما چگونه می‌توانید شروع به تفکر در مورد استیت‌ها کنید؟ اکسترکت کردن متدهای معنادار از کدهای موجود شروع خوبی است، اما فقط آغاز راه است. مبنا این است که ماشین‌های استیت را درک کنیم. شاید خاطرات بدی از درس‌های علوم کامپیوتر داشته باشید، اما سعی کنید آن‌ها را کنار بگذارید. ماشین‌های استیت ذاتاً پیچیده نیستند. سعی کنید آن‌ها را به‌صورت تصویری تجسم کنید تا درکشان آسان‌تر شود. کد خود را با تست هدایت کنید تا وضعیت‌ها و انتقال‌های معتبر و نامعتبر را رمزگشایی کرده و آن‌ها را درست نگه دارید. الگوی استیت را مطالعه کنید. وقتی آن را درک کردید، در مورد طراحی با قرارداد[[12]](#footnote-12) بخوانید. این به شما کمک می‌کند تا با اعتبارسنجی داده‌های ورودی و خود شیء در ورود و خروج هر متد عمومی، یک استیت معتبر را تضمین کنید.

اگر استیت شما نادرست باشد، یک باگ وجود دارد و اگر لغو نکنید، خطر ازبین‌رفتن داده‌ها را دارید. اگر بررسی‌های استیت را مزاحمت بدانید، یاد بگیرید چگونه از یک ابزار استفاده کنید، کد تولید کنید، کدها را کنار هم قرار دهید یا از ویژگی‌ها برای پنهان‌کردن آنها استفاده کنید. صرف‌نظر از اینکه کدام رویکرد را انتخاب کنید، تفکر در مورد استیت‌ها کد شما را ساده‌تر و مقاوم‌تر خواهد کرد.

**دو فکر بهتر از یک فکر است**

*آدریان ویبل*

برنامه‌نویسی نیازمند تفکر عمیق است و تفکر عمیق نیازمند تنهایی. این تصویر کلیشه‌ای از برنامه‌نویس است.

این رویکرد «گرگ تنها» در برنامه‌نویسی جای خود را به رویکردی مشارکتی‌تر می‌دهد که به باور من کیفیت، بهره‌وری و رضایت شغلی برنامه‌نویسان را ارتقا می‌دهد. این رویکرد توسعه‌دهندگان را وادار می‌کند تا با یکدیگر و همچنین با غیربرنامه‌نویسان- تحلیلگران کسب‌وکار و سیستم، متخصصان کنترل کیفیت و کاربران بیشتر همکاری کنند.

این برای برنامه‌نویسان چه معنایی دارد؟ دیگر تنها متخصص فناوری بودن کافی نیست. شما باید در کارگروهی مؤثر واقع شوید.

همکاری صرفاً درباره پرسیدن و پاسخ‌دادن به سؤالات یا حضور در جلسات نیست. همکاری درباره بالازدن آستین‌ها به همراه شخص دیگری برای حمله مشترک به کار است.

من شدیداً طرفدار برنامه‌نویسی جفتی هستم. شاید بتوان آن را «همکاری حداکثری» نامید. به‌عنوان یک توسعه‌دهنده، مهارت‌های من هنگام کار جفتی رشد می‌کند. اگر در حوزه یا فناوری ضعیف‌تر از شریکم باشم، واضح است که از تجربیات او استفاده می‌کنم و چیزهای جدیدی یاد می‌گیرم. وقتی در بعضی جنبه‌ها قوی‌ترم، با مجبورشدن به توضیح‌دادن، بیشتر درباره آنچه می‌دانم و نمی‌دانم یاد می‌گیرم. مسلماً، هر دوی ما چیزهای جدیدی بلدیم و از یکدیگر یاد می‌گیریم.

هنگام کار جفتی، هر یک از ما تجربیات برنامه‌نویسی انباشته خود را- در حوزه و همچنین فنی- به مسئله موردنظر می‌آوریم و می‌توانیم بینش و تجربه منحصربه‌فردی را در نوشتن نرم‌افزار به‌صورت مؤثر و کارآمد ارائه دهیم. حتی در موارد عدم تعادل شدید در دانش حوزه یا فنی، شرکت‌کننده باتجربه‌تر قطعاً چیزی از دیگری یاد می‌گیرد- شاید یک میان‌بر صفحه‌کلید جدید یا آشنایی با ابزار یا کتابخانه‌ای جدید. برای عضو کم‌تجربه‌تر جفت، این راه عالی برای پیشرفت سریع است.

برنامه‌نویسی جفتی در میان طرفداران توسعه نرم‌افزار چابک، هرچند نه انحصاراً، اما محبوب است. بعضی از مخالفان کار جفتی می‌پرسند «چرا باید برای کار یک نفره دو برنامه‌نویس را استخدام کنم؟» پاسخ من این است که بله، شما نباید این کار را کنید. من استدلال می‌کنم که کار جفتی کیفیت، درک از حوزه و فناوری و تکنیک‌ها (مانند ترفندهای IDE) را افزایش می‌دهد و اثر ریسک قرعه‌کشی (یکی از توسعه‌دهندگان متخصص شما برنده قرعه‌کشی می‌شود و روز بعد استعفا می‌دهد) را کاهش می‌دهد.

ارزش بلندمدت یادگیری یک میانبر صفحه‌کلید جدید چیست؟ چگونه بهبود کلی کیفیت محصول ناشی از کار جفتی را اندازه‌گیری می‌کنیم؟ چطور اثر شریکتان را که اجازه نمی‌دهد دنبال رویکردی بی‌فایده برای حل یک مسئله دشوار بروید، اندازه می‌گیریم؟ یک مطالعه افزایش 40% در اثربخشی و سرعت را نشان می‌دهد.\* ارزش کاهش «ریسک قرعه‌کشی» شما چقدر است؟ اندازه‌گیری اکثر این مزایا دشوار است.

چه کسی باید با چه کسی به‌صورت جفتی کار کند؟ اگر عضو جدید تیم هستید، پیداکردن عضوی از تیم که دانش کافی داشته باشد مهم است. پیداکردن کسی که مهارت‌های بین‌فردی و مربیگری خوبی دارد نیز همان قدر مهم است. اگر تجربه زیادی در حوزه ندارید، با عضوی از تیم که در آن حوزه متخصص است همکاری کنید.

اگر متقاعد نشده‌اید، آزمایش کنید: با همکارانتان همکاری نمایید. در مورد یک مسئله جالب و پیچیده به‌صورت جفتی کار کنید. ببینید چه حسی دارد. چند بار این کار را امتحان کنید.

**دو اشتباه می‌توانند درست باشند (و رفع آنها دشوار است)**

*آلن کِلی*

کد هرگز دروغ نمی‌گوید، اما می‌تواند با خودش تناقض داشته باشد. برخی تناقضات منجر به بروز این سؤال می‌شوند که «چطور ممکن است این کد کار کند؟»

در یک مصاحبه،[[13]](#footnote-13)\* طراح اصلی نرم‌افزار ماژول ماه‌نشین آپولو 11، آلن کلامپ، افشا کرد که نرم‌افزار کنترل‌کننده موتورها حاوی یک باگ بود که باعث ناپایداری فرود آور می‌شد. بااین‌حال، یک باگ دیگر باعث جبران باگ اول شده بود و نرم‌افزار قبل از پیداکردن یا اصلاح هر باگی، برای فرودهای ماه آپولو 11 و 12 استفاده شد.

حالا یک تابع بازگشت وضعیت تکمیل را در نظر بگیرید. تصور کنید وقتی باید مقدار true برگرداند، مقدار false را برمی‌گرداند. حالا تصور کنید تابع فراخواننده فراموش کرده بازگشتی را چک کند. همه‌چیز به‌خوبی کار می‌کند تا اینکه یک روز کسی چک ازدست‌رفته را متوجه می‌شود و آن را اضافه می‌کند.

یا در نظر بگیرید یک برنامه که وضعیت را به‌عنوان یک سند XML ذخیره می‌کند. تصور کنید یکی از گره‌ها به‌جای TimeToDie که در مستندات گفته بود، اشتباهاً به‌عنوان TimeToLive نوشته شده است. تا زمانی که کدهای نویسنده و خواننده هر دو حاوی همان خطا باشند، همه‌چیز ظاهراً درست به نظر می‌رسد. اما با اصلاح یکی یا اضافه‌کردن برنامه جدیدی که همان سند را می‌خواند، تقارن و همراه آن کد از بین می‌رود.

وقتی دو عیب در کد یک خطای قابل‌مشاهده ایجاد کنند، رویکرد منظم برای اصلاح خطاها نیز می‌تواند فروبپاشد. توسعه‌دهنده گزارش خطا دریافت می‌کند، نقص را پیدا می‌کند، آن را اصلاح می‌کند و دوباره تست می‌کند. بااین‌حال، خطای گزارش شده همچنان رخ می‌دهد، زیرا یک نقص دوم در کار است؛ بنابراین اصلاح اولیه برگردانده می‌شود، کد بررسی می‌شود تا زمانی که نقص پایه دوم پیدا شود و برای آن اصلاحی اعمال می‌شود. اما نقص اول برگشته است، خطای گزارش شده همچنان دیده می‌شود، و بنابراین اصلاح دوم لغو می‌شود. این فرایند تکرار می‌شود، اما حالا توسعه‌دهنده دو اصلاح احتمالی را رد کرده و در حال جستجو برای اصلاحی سوم است که هرگز کار نخواهد کرد.

تعامل بین دو عیب کدگذاری که به‌عنوان یک خطای قابل‌مشاهده تکی ظاهر می‌شوند، نه‌تنها اصلاح مشکل را سخت می‌کند، بلکه باعث می‌شود توسعه‌دهندگان به کوره‌راه‌هایی بروند و تنها متوجه شوند که در ابتدا، پاسخ‌های درست را امتحان کرده‌اند.

این مسئله فقط در کد اتفاق نمی‌افتد: مشکل همچنین در مستندات نیازمندی‌های مکتوب وجود دارد و می‌تواند به‌صورت ویروسی از جایی به‌جای دیگر گسترش یابد. یک خطا در کد یک خطا در توصیف مکتوب را جبران می‌کند.

می‌تواند به افراد نیز سرایت کند: کاربران می‌آموزند که وقتی برنامه می‌گوید چپ، منظورش راست است، پس رفتار خود را به‌تناسب تنظیم می‌کنند. حتی این را به کاربران جدید منتقل می‌کنند: «یادت باشد وقتی برنامه می‌گوید «روی دکمه چپ کلیک کن»، واقعاً منظورش دکمه سمت راست است.» اصلاح باگ، و ناگهان کاربران نیاز به آموزش مجدد دارند.

اشتباهات تکی می‌توانند آسان به چشم آیند و آسان اصلاح شوند. مسائلی که علل چندگانه دارند و نیازمند چندین تغییر هستند، حل آن‌ها سخت‌تر است. تا حدی، این بدان خاطر است که رفع مسائل آسان به قدری سریع و آسان است که مردم تمایل دارند آن‌ها را نسبتاً سریع رفع کنند و مسائل دشوارتر را برای تاریخی دیرتر ذخیره کنند.

توصیه ساده‌ای برای مقابله با خطاهای ناشی از نقص‌های همدل وجود ندارد. برای رسیدگی به این مسائل، آگاهی از احتمال وقوع، فکری روشن و تمایل به درنظرگرفتن همه احتمالات ضروری است.

**برنامه‌نویسی اوبونتو برای دوستانتان**

*اَسلم خان*

اغلب، کد را به‌صورت انزوا نوشته و آن کد منعکس‌کننده تفسیر شخصی ما از یک مسئله و همچنین یک راه‌حل بسیار شخصی شده است. ممکن است بخشی از تیم باشیم، اما انزوا هستیم، همان‌طور که تیم نیز انزواست. بسیار سریع فراموش می‌کنیم که این کد ایجاد شده در انزوا توسط دیگران اجرا، استفاده، گسترش و متکی خواهد شد. جنبه اجتماعی خلق نرم‌افزار را آسان نادیده می‌گیریم. ایجاد نرم‌افزار، ترکیبی از تمرین فنی و تمرین اجتماعی است. فقط باید سرمان را بیشتر بلند کنیم و متوجه شویم که در انزوا کار نمی‌کنیم و مسئولیت مشترکی برای افزایش احتمال موفقیت همه، نه فقط تیم توسعه، داریم.

می‌توانید کد باکیفیت خوبی را در انزوا بنویسید، درحالی‌که کاملاً در خود گم شده‌اید. از یک دیدگاه، این رویکردی خودمحورانه است (نه خودخواهانه، بلکه شخصی). این همچنین دیدگاه فلسفی ذن است و در مورد شما در آن لحظه خلق کد است. همیشه سعی می‌کنم در لحظه زندگی کنم؛ چون به من کمک می‌کند به کیفیت خوب نزدیک شوم، اما سپس در لحظه خودم زندگی می‌کنم. ولی لحظه تیم من چطور؟ آیا لحظه من همان لحظه تیم است؟

در زبان زولو، فلسفه اوبونتو به‌عنوان «Umuntu ngumuntu ngabantu» خلاصه می‌شود که تقریباً به معنای «یک فرد از طریق (سایر) افراد فرد است» است. من بهتر می‌شوم؛ چون تو با اعمال خوبت مرا بهتر می‌کنی. وجه مقابل این است که تو در کارت بدتر می‌شوی وقتی من در کارم بد عمل می‌کنم.

طرف دیگر این است که شما در کاری که انجام می‌دهید بدتر می‌شوید وقتی من در کاری که انجام می‌دهم بد باشم. در میان توسعه‌دهندگان، می‌توانیم آن را به «یک توسعه‌دهنده یک توسعه‌دهنده از طریق (سایر) توسعه‌دهندگان» محدود کنیم. اگر آن را به فلز پایین بیاوریم، «کد از طریق کد (سایر) کد است».

در میان توسعه‌دهندگان، می‌توانیم آن را به «یک توسعه‌دهنده از طریق (سایر) توسعه‌دهندگان توسعه‌دهنده است» تقلیل دهیم. اگر آن را به پایین‌ترین سطح برسانیم، آنگاه «یک کد از طریق (سایر) کدها کد است».

کیفیت کدی که می‌نویسم، بر کیفیت کدی که تو می‌نویسی تأثیر می‌گذارد. اگر کد من کیفیت پایینی داشته باشد چه؟ حتی اگر شما کد بسیار تمیزی بنویسید، جایی که از کد من استفاده می‌کنید، کیفیت کد شما تا حدودی به کیفیت کد من تنزل خواهد یافت. می‌توانید الگوها و تکنیک‌های زیادی را برای محدودکردن آسیب اعمال کنید، اما آسیب قبلاً وارد شده است. من باعث شده‌ام شما کار بیشتری از آنچه نیاز داشتید انجام دهید، فقط به این دلیل ساده که من هنگام زندگی در لحظه خودم، به شما فکر نکردم.

ممکن است کد خود را تمیز بدانم، اما هنوز می‌توانم آن را با برنامه‌نویسی اوبونتو بهتر کنم. کد اوبونتو به چه شکل است؟ دقیقاً مثل کد خوب و تمیز است. مسئله کد به‌عنوان محصول نیست. مسئله، عمل ایجاد آن محصول است. برنامه‌نویسی برای دوستانتان، با اوبونتو، به تیمتان کمک می‌کند ارزش‌هایتان را زندگی کنید و اصولتان را تقویت نمایید. نفر بعدی که به هر شکلی با کد شما سروکار دارد، فرد بهتر و توسعه‌دهنده بهتری خواهد بود.

فلسفه ذن درباره فرد است. اوبونتو درباره ذن برای گروهی از مردم است. بسیار به‌ندرت کدی برای تنها خودمان می‌نویسیم.

**ابزارهای یونیکس دوستان شما هستند**

*دیومیدیس اسپینلیس*

اگر در مسیر تبعید به یک جزیره بیابانی، مجبور شوم بین یک IDE و جعبه ابزار یونیکس یکی را انتخاب کنم، بدون تردید ابزارهای یونیکس را انتخاب می‌کنم. دلایلی که باید در ابزارهای یونیکس مهارت پیدا کنید عبارت‌اند از:

اولاً، IDEها برای زبان‌های خاص هدف‌گذاری می‌شوند، درحالی‌که ابزارهای یونیکس می‌توانند با هر چیزی که به شکل متنی ظاهر می‌شود کار کنند. در محیط توسعه امروزی که زبان‌ها و نمادگذاری‌های جدید هر ساله ظهور می‌کنند، یادگیری کار به روش یونیکس سرمایه‌گذاری است که بارهاوبارها جواب خواهد داد.

علاوه بر این، درحالی‌که IDEها فقط دستوراتی که توسعه‌دهندگانشان طراحی کرده‌اند را ارائه می‌دهند، با ابزارهای یونیکس می‌توانید هر کاری را که می‌توانید تصور کنید انجام دهید. به این ابزار به‌عنوان قطعات لگوی کلاسیک (پیش از بیونیکل) فکر کنید: می‌توانید با ترکیب ابزارهای کوچک اما چندمنظوره یونیکس، دستورات خود را ایجاد کنید. به‌عنوان‌مثال، توالی زیر یک پیاده‌سازی متنی از تحلیل امضای کانینگهام است- توالی نقطه‌ویرگول‌ها، آکولادها و نقل‌قول‌های هر فایل که می‌تواند درباره محتوای فایل چیزهای زیادی بگوید:

for i in \*.java; do

echo -n "$i: "

sed 's/[^"{};]//g' $i | tr -d '\n'

echo

done

علاوه بر این، هر عملیاتی که در یک IDE یاد بگیرید، مختص به آن کار مشخص است- به‌عنوان مثال، اضافه کردن یک مرحله جدید در پیکربندی ساخت دیباگ یک پروژه. در مقابل، تقویت مهارت‌هایتان در ابزار یونیکس شما را در هر کاری مؤثرتر می‌کند. به‌عنوان مثال، از ابزار sed که در توالی دستور قبلی استفاده شد، برای تبدیل ساخت یک پروژه جهت کامپایل متقاطع بر روی چندین ساختار معماری پردازنده استفاده کردم.

ابزارهای یونیکس در عصری توسعه یافتند که یک رایانه چندکاربره تنها ۱۲۸ ‎کیلوبایت حافظه رم داشت. ذکاوتی که در طراحی آن‌ها به کار رفته است باعث می‌شود در دنیای امروز بتوانند بسیار کارآمد با مجموعه‌داده‌های عظیم کار کنند. اکثر این ابزارها مانند فیلتر عمل می‌کنند و فقط یک خط را در یک زمان پردازش می‌کنند، به این معنی که محدودیت بالایی در مقدار داده‌ای که می‌توانند مدیریت کنند وجود ندارد. می‌خواهید برای تعداد ویرایش‌های ذخیره شده در دامپ نیم ترابایتی ویکی‌پدیا انگلیسی جستجو کنید؟ یک فراخوانی ساده

grep '<revision>' | wc –l

پاسخ را بدون هیچ دردسری به شما می‌دهد. اگر یک توالی دستور عموماً مفید پیدا کردید، می‌توانید آن را به‌راحتی در یک شِل اسکریپت بسته‌بندی کنید و از سازه‌های برنامه‌نویسی به طور منحصربه‌فرد قدرتمندی استفاده کنید، مانند انتقال داده به درون حلقه‌ها و شرطی‌ها. حتی تأثیرگذارتر اینکه دستورات یونیکس که به‌صورت لوله‌کشی اجرا می‌شوند، مثل مثال قبلی، به طور طبیعی بارشان را بین واحدهای پردازش متعدد CPUهای امروزی چندهسته‌ای توزیع می‌کنند.

منشأ کوچک و زیبا و پیاده‌سازی‌های اوپن سورس ابزار یونیکس باعث می‌شود آن‌ها به طور فراگیر حتی بر روی پلتفرم‌های محدود در منابع نیز در دسترس باشند، مانند پخش‌کننده رسانه‌ای یا مسیریاب DSL من. احتمال اینکه چنین دستگاه‌هایی رابط کاربری گرافیکی قدرتمندی ارائه دهند بعید است، اما اغلب شامل برنامه BusyBox است که پرکاربردترین ابزار را فراهم می‌کند و اگر شما بر روی ویندوز در حال توسعه هستید، محیط Cygwin تمام ابزار یونیکس قابل‌تصور، هم به‌صورت اجرایی و هم در فرم سورس کد را در اختیار شما قرار می‌دهد.

در نهایت، اگر هیچ‌کدام از ابزارهای موجود با نیازهای شما تطابق نداشت، گسترش دنیای ابزارهای یونیکس خیلی آسان است. فقط یک برنامه (به هر زبان موردعلاقه‌تان) بنویسید که به چند قاعده ساده پایبند باشد: برنامه شما باید تنها یک کار انجام دهد؛ باید داده را به‌عنوان خطوط متنی از ورودی استاندارد خود بخواند؛ و باید نتایج خود را بدون سربرگ و سایر مزاحمت‌ها روی خروجی استاندارد خود نمایش دهد. پارامترهای مؤثر بر عملکرد ابزار در command line داده می‌شوند. این قوانین را دنبال کنید و «زمین و هر آنچه در آن است متعلق به شماست.»

**از الگوریتم و ساختار داده مناسب استفاده کنید**

*ژان کریستیان «جی سی» ون وینکل*

*یک بانک بزرگ با شعبات زیاد شکایت کرد که کامپیوترهای جدیدی که برای صندوق‌دارها خریده بود، بسیار کند هستند. این در زمانی بود که هنوز همه از بانکداری الکترونیکی استفاده نمی‌کردند، و خودپردازها نیز همانند امروز رواج نداشتند. مردم خیلی بیشتر به بانک‌ها مراجعه می‌کردند و کامپیوترهای کند منجر به صف‌بندی مردم می‌شدند. در نتیجه، بانک تهدید کرد قراردادش با تأمین‌کننده را لغو خواهد کرد.*

*تأمین‌کننده یک متخصص تحلیل و بهینه‌سازی عملکرد را برای تعیین علت تأخیر اعزام کرد. او خیلی زود یک برنامه خاص را که روی ترمینال اجرا می‌شد، یافت که تقریباً تمام ظرفیت CPU را مصرف می‌کرد. با استفاده از ابزار پروفایلینگ، برنامه را بزرگ‌نمایی کرد و توانست تابعی را که مقصر بود ببیند. سورس کد شامل این کد بود:*

for (i=0; i<strlen(s); ++i) {

if (... s[i] ...) ...

}

*و رشته s به طور متوسط شامل هزاران کاراکتر بود. کد (نوشته شده توسط بانک) به‌سرعت تغییر کرد، و صندوق‌دارهای بانک بعد از آن به خوبی و خوشی زندگی کردند...*

آیا برنامه‌نویس نباید به‌جای استفاده از کدی که بی‌دلیل با مقیاس کوادراتیک کار می‌کند، بهتر عمل می‌کرد؟

هر فراخوانی به strlen هر یک از هزاران کاراکتر بسیار زیاد استرینگ را برای یافتن کاراکتر null خاتمه‌دهنده آن مرور می‌کرد. بااین‌حال، استرینگ هرگز تغییر نمی‌کرد. با تعیین طول آن از پیش، برنامه‌نویس می‌توانست از هزاران فراخوانی به strlen (و میلیون‌ها اجرای حلقه) جلوگیری کند:

n=strlen(s);

for (i=0; i<n; ++i) {

if (... s[i] ...) ...

}

همه ضرب‌المثل «ابتدا آن را طوری بساز که کار کند، بعد کاری کن تا سریع کار کند» را می‌شناسند تا از تله بهینه‌سازی میکرو جلوگیری کنند. اما مثال قبل تقریباً باعث می‌شود باور کنید که برنامه‌نویس ضرب‌المثل ماکیاولی را دنبال می‌کرد «ابتدا آن را طوری بساز که کند کار کند».

این بی‌تفاوتی چیزی است که ممکن است بیش از یک بار با آن مواجه شوید و فقط یک مسئله «دوباره چرخ اختراع نکن» نیست. گاهی برنامه‌نویسان تازه‌کار فقط شروع به تایپ می‌کنند بدون اینکه واقعاً فکر کنند، و ناگهان «مرتب‌سازی حبابی» را «اختراع» می‌کنند. ممکن است ممکن است حتی در مورد آن لاف بزنند.

طرف دیگر انتخاب الگوریتم درست، انتخاب ساختمان داده است. این انتخاب می‌تواند تفاوت بزرگی ایجاد کند: استفاده از یک لیست پیوندی برای مجموعه‌ای شامل یک میلیون آیتم که می‌خواهید جستجو کنید - در مقایسه با یک ساختمان داده هش شده یا درخت باینری - تأثیر زیادی بر ارزیابی کاربر از برنامه‌نویسی شما خواهد گذاشت.

برنامه‌نویسان نباید چرخ را دوباره اختراع کنند، و باید در صورت امکان از کتابخانه‌های موجود استفاده کنند. اما برای توانایی جلوگیری از مشکلاتی مانند مشکل بانک، باید در مورد الگوریتم‌ها و نحوه مقیاس‌بندی آنها نیز آموزش ببینند. آیا فقط جاذبه بصری در ویرایشگرهای متنی مدرن است که آن­ها را به کندی برنامه‌های قدیمی مانند WordStar در دهه 1980 می‌کند؟ بسیاری می‌گویند استفاده مجدد در برنامه‌نویسی از اهمیت والایی برخوردار است. بااین‌حال، برنامه‌نویسان باید بدانند چه زمانی، چگونه و از چه چیزی استفاده مجدد کنند. برای انجام این کار، آنها باید دانش حوزه مسئله و الگوریتم‌ها و ساختمان داده‌ها را داشته باشند.

همچنین یک برنامه‌نویس خوب باید بداند چه زمانی از یک الگوریتم نامطلوب استفاده کند. به‌عنوان مثال اگر حوزه مسئله تعیین کند که هرگز نمی‌تواند بیش از پنج آیتم (مانند تعداد تاس‌ها در بازی یاتزی) باشد، می‌دانید که همیشه حداکثر پنج آیتم را باید مرتب کنید. در این صورت، مرتب‌سازی حبابی در واقع ممکن است کارامدترین روش برای مرتب کردن آیتم ها باشد. هر سگی روز خوشی دارد.

بنابراین، چند کتاب خوب بخوانید و مطمئن شوید که آنها را درک کرده‌اید و اگر واقعاً کتاب «هنر برنامه‌نویسی رایانه‌ای» (Addison-Wesley Professional) از دونالد کنوت را بخوانید، خب، حتی ممکن است شانس بیاورید: یک اشتباه توسط نویسنده پیدا کنید، و یکی از چک‌های دلاری هگزادسیمال دونالد کنوت (2.56 دلار) را به دست بیاورید.

**ورود به سیستم با جزئیات می‌تواند خواب شما را مختل کند**

*یوهانس برادوال*

وقتی با سیستمی که برای مدتی درحال‌توسعه یا تولید بوده است مواجه می‌شوم، اولین نشانه مشکل واقعی همیشه یک ورود به سیستم پر مسئله است. می‌دانید درباره چه صحبت می‌کنم: وقتی کلیک‌کردن روی لینکی در یک جریان عادی در یک صفحه وب منجر به طوفانی از پیام‌ها در تنها لاگی می‌شود که سیستم فراهم می‌کند. ورود بیش از حد به سیستم می‌تواند؛ مانند عدم ورود به سیستم، بی‌فایده باشد.

اگر سیستم‌های شما مثل سیستم‌های من باشند، وقتی کار شما تمام شد، کار کس دیگری تازه شروع می‌شود. پس از توسعه سیستم، امیدواریم آن سیستم طولانی و پربار زندگی کند و به مشتریان خدمت کند (اگر خوش‌شانس باشید). چطور متوجه می‌شوید اگر هنگامی که سیستم در حال تولید است چیزی اشتباه پیش برود و چطور با آن مقابله می‌کنید؟

شاید کسی سیستم شما را نظارت کند یا شاید خودتان به آن نظارت کنید. درهرصورت، احتمالاً لاگ‌ها بخشی از نظارت خواهند بود. اگر چیزی نمایان شود و مجبور شوید بیدار شوید تا با آن مقابله کنید، می‌خواهید مطمئن شوید دلیل خوبی برای آن وجود دارد. اگر سیستم من در حال مرگ است، می‌خواهم بدانم. اما اگر فقط یک لرزش کوچک باشد، ترجیح می‌دهم از خواب زیبایم لذت ببرم.

برای بسیاری از سیستم‌ها، اولین نشانه اینکه چیزی اشتباه است، یک پیام لاگ است که در برخی از گزارش‌ها نوشته می‌شود. اغلب، این پیام ارور لاگ خواهد بود؛ بنابراین لطفاً از روز اول مطمئن شوید که اگر چیزی در ارور لاگ ثبت شد، حاضر هستید کسی در میانه شب با شما تماس بگیرد و بیدارتان کند. اگر بتوانید در تست سیستم، باری شبیه به بار واقعی شبیه‌سازی کنید، نبود پیام خطا در لاگ اولین نشانه خوب پایداری سیستم است - یا هشدار اولیه‌ای مبنی بر عدم پایداری سیستم.

سیستم‌های غیرمتمرکز سطح دیگری از پیچیدگی را اضافه می‌کنند. باید تصمیم بگیرید چگونه با شکست وابستگی خارجی مقابله کنید. اگر سیستم شما بسیار غیرمتمرکز باشد، این ممکن است یک رویداد معمول باشد. مطمئن شوید سیاست ورود به سیستم شما این موضوع را در نظر می‌گیرد.

به‌طورکلی، بهترین نشانه اینکه همه چیز درست پیش می‌رود، حضور پیام‌های با اولویت پایین‌تر است. من انتظار دارم به‌ازای هر رویداد مهم برنامه‌نویسی، حدود یک پیام INFO-level در سیستم ثبت شود.

یک سیستم پرسروصدا نشان می‌دهد کنترل آن در محیط تولید دشوار خواهد بود. اگر انتظار لاگ ارور نداشته باشید، آسان‌تر می‌توانید زمانی که رخ می‌دهد، با آن برخورد کنید.

**WET کاهش عملکرد گلوگاه‌ها را رقیق می‌کند**

*کرک پپردین*

اهمیت اصل DRY (خودتان را تکرار نکنید) این است که هر دانشی در یک سیستم باید تنها یک بار نمایش داده شود. به‌عبارت‌دیگر، دانش باید در یک پیاده‌سازی منحصربه‌فرد قرار بگیرد. نقطه مقابل DRY، WET (هر بار بنویسید) است. کد ما زمانی WET است در آن دانش در چند پیاده‌سازی جداگانه کدنویسی می‌شود. تفاوت عملکردی DRY و WET وقتی مشخص می‌شود که اثر آنها بر یک پروفایل عملکردی را در نظر بگیریم.

اجازه دهید از یک ویژگی سیستم، مثلاً X که یک گلوگاه CPU است شروع کنیم. فرض کنیم ویژگی X، 30% از CPU را مصرف می‌کند. حالا فرض کنید که ویژگی X، 10 پیاده‌سازی مختلف دارد. به طور میانگین، هر پیاده‌سازی ٪3 از CPU را مصرف خواهد کرد. از آنجایی که این سطح از استفاده از CPU ارزش نگرانی را ندارد اگر به دنبال یک پیروزی سریع هستیم، احتمالاً از دست خواهیم داد که این ویژگی گلوگاه ماست. با این حال، فرض کنید که ما به نحوی ویژگی X را به‌عنوان یک گلوگاه شناسایی کرده‌ایم. اکنون ما با مشکل پیدا کردن و اصلاح هر پیاده‌سازی منفرد روبرو هستیم. با WET، ما 10 پیاده‌سازی مختلف داریم که باید پیدا و اصلاح کنیم. با DRY، ما به وضوح مصرف ٪30 CPU را می‌دیدیم و یک دهم کد را برای اصلاح داشتیم. بنابراین شناسایی و تعمیر آسان‌تر است.

در یک مورد استفاده، ما اغلب مرتکب نقض DRY در استفاده از مجموعه‌ها (کلکسیون‌ها) می‌شویم. یک تکنیک رایج برای پیاده‌سازی یک درخواست، تکرار بر روی مجموعه و اعمال درخواست به هر عنصر است:

public class UsageExample {

private ArrayList<Customer> allCustomers = new ArrayList<Customer>();

// ...

public ArrayList<Customer> findCustomersThatSpendAtLeast(Money amount) {

ArrayList<Customer> customersOfInterest = new ArrayList<Customer>();

for (Customer customer: allCustomers) {

if (customer.spendsAtLeast(amount))

customersOfInterest.add(customer);

}

return customersOfInterest;

}

}

با نمایش مستقیم مجموعه داده خام به مشتریان، اصل کپسوله‌سازی را نقض کرده‌ایم. این نه‌تنها توانایی ما برای رفکتور کردن را محدود می‌کند، بلکه مشتریان را مجبور می‌کند تا پرس‌وجوهای مشابه را مجدداً پیاده‌سازی کنند که نقض DRY است. این مشکل به‌راحتی با حذف مجموعه خام از API و معرفی یک نوع داده جدید به نام CustomerList قابل‌حل است. این کلاس جدید بیشتر با دامنه مسئله همخوانی دارد و می‌تواند خانه طبیعی برای تمام درخواست‌ها باشد.

داشتن این نوع مجموعه جدید به ما اجازه می‌دهد به‌راحتی ببینیم آیا این درخواست‌ها گلوگاه عملکرد هستند یا خیر. با گنجاندن پرس‌و‌جوها در کلاس، نیازی به نمایش گزینه‌های بازنمایی، مانند ArrayList، برای مشتریان خود را حذف می‌کنیم. این به ما این آزادی را می‌دهد که بدون ترس از نقض قراردادهای مشتری، این پیاده‌سازی‌ها را تغییر دهیم:

public class CustomerList {

private ArrayList<Customer> customers = new ArrayList<Customer>();

private SortedList<Customer> customersSortedBySpendingLevel =

new SortedList<Customer)();

// ...

public CustomerList findCustomersThatSpendAtLeast(Money amount) {

return new CustomerList(

customersSortedBySpendingLevel.elementsLargerThan(amount));

}

}

public class UsageExample {

public static void main(String[] args) {

CustomerList customers = new CustomerList();

// ...

CustomerList customersOfInterest =

customers.findCustomersThatSpendAtLeast(someMinimalAmount);

// ...

}

}

مهم‌تر از جزئیاتِ این مثال خاص، پیروی از DRY به ما کمک کرد تا یک گلوگاه عملکردی را شناسایی و رفع کنیم که در صورتِ WET بودنِ کد، یافتن آن دشوارتر می‌شد.

**زمانی که برنامه‌نویس‌ها و تسترها شروع به همکاری می‌کنند**

*جنت گرگوری*

زمانی که تسترها و برنامه‌نویس‌ها شروع به همکاری می‌کنند، اتفاق جادویی رخ می‌دهد. زمان کمتری صرف ارسال باگ‌ها به جلو و عقب از طریق سیستم ردیابی نقص می‌شود. زمان کمتری صرف ارسال باگ‌ها به جلو و عقب و تعیین اینکه آیا یک مورد واقعاً باگ است یا ویژگی جدید، می‌شود. به جای آن، زمان بیشتری صرف توسعه نرم‌افزار خوب برای برآورده‌کردن نیازهای مشتری می‌شود. فرصت‌های زیادی برای شروع همکاری قبل از شروع کدنویسی وجود دارد.

تسترها می‌توانند به مشتریان در نوشتن و خودکارسازی تست‌های پذیرش با استفاده از زبان دامنه آنها و ابزارهایی مانند Fit[[14]](#footnote-14) کمک کنند. زمانی که این تست‌ها قبل از شروع کدنویسی به برنامه‌نویسان داده می‌شود، تیم در حال تمرین توسعه محور تست پذیرش (ATDD)[[15]](#footnote-15) است. برنامه‌نویسان فیکسچرهایی برای اجرای تست‌ها می‌نویسند، و سپس کد می‌نویسند تا تست‌ها را پاس کنند. این تست‌ها سپس بخشی از مجموعه رگرسیون می‌شوند. در نتیجه تست‌های عملکردی زودتر کامل می‌شوند و زمان بیشتری برای تست اکتشافی باقی می‌ماند.

تسترها می‌توانند با نوشتن و خودکارسازی تست‌های پذیرش با استفاده از زبان دامنه و ابزارها، قبل از شروع کدنویسی به برنامه‌نویسان کمک کنند. برنامه‌نویسان فیکسچرهایی برای اجرای تست‌ها می‌نویسند و سپس کد را برای پاس‌کردن آن‌ها می‌نویسند. این تست‌ها بخشی از تست‌های رگرسیونی می‌شوند. در نتیجه تست‌های عملکردی زودتر کامل می‌شوند و زمان بیشتری برای تست اکتشافی باقی می‌ماند.

می‌توانیم یک گام فراتر برویم. به‌عنوان یک تستر، می‌توانم بیشتر ایده‌های تست خود را قبل از شروع کدنویسی یک ویژگی جدید توسط برنامه‌نویسان ارائه دهم. وقتی از برنامه‌نویسان می‌پرسم آیا پیشنهادی دارند، تقریباً همیشه به من اطلاعاتی می‌دهند که به من در پوشش بهتر تست کمک می‌کند، یا به من کمک می‌کند از صرف زمان زیاد برای تست‌های غیرضروری جلوگیری کنم. اغلب، از بروز نقص‌ها جلوگیری کرده‌ایم؛ زیرا تست‌ها بسیاری از ایده‌های اولیه را روشن می‌کنند. برای مثال، در یک پروژه‌ای که در آن بودم، تست‌های Fit که به برنامه‌نویسان دادم نتایج مورد انتظار یک جستجوی جایگزین را نشان می‌داد. برنامه‌نویس قصد داشت فقط جستجوهای کلمات کامل را کدگذاری کند. ما توانستیم قبل از شروع کدنویسی با مشتری صحبت کنیم و تفسیر صحیح را تعیین کنیم. با همکاری، از بروز نقص جلوگیری کردیم و باعث شد زمان زیادی از هر دوی ما ذخیره شود.

برنامه‌نویسان نیز می‌توانند با تسترها برای ایجاد یک اتوماسیون موفق همکاری کنند. برنامه‌نویسان اصول کدنویسی خوب را درک می‌کنند و می‌توانند به تسترها در تنظیم یک مجموعه اتوماسیون تست قابل‌اطمینان که برای کل تیم کار می‌کند، کمک کنند. اغلب دیده‌ام که پروژه‌های اتوماسیون تست به دلیل طراحی ضعیف تست‌ها شکست می‌خورند. تست‌ها سعی می‌کنند خیلی زیاد تست کنند، یا تسترها درک کافی از فناوری برای نگه‌داشتن مستقل تست‌ها ندارند. تسترها اغلب گلوگاه هستند، بنابراین برای برنامه‌نویسان منطقی است که در تکالیفی مانند اتوماسیون با آن‌ها همکاری کنند. همکاری با تسترها برای درک اینکه چه چیزهایی را می‌توان از ابتدا تست کرد، شاید با ارائه یک ابزار ساده، به برنامه‌نویسان یک چرخه دیگر از بازخورد خواهد داد که به آن‌ها در تحویل کد بهتر در درازمدت کمک خواهد کرد.

زمانی که تسترها دیگر فکر نمی‌کنند که تنها شغل آن‌ها شکستن نرم‌افزار و پیداکردن باگ در کد برنامه‌نویس‌ها است، برنامه‌نویسان دیگر فکر نمی‌کنند که تسترها «در پی آن‌ها هستند» و بیشتر به همکاری تمایل پیدا می‌کنند. وقتی برنامه‌نویسان متوجه می‌شوند که مسئول ساخت کیفیت در کد خود هستند، قابلیت تست کد یک محصول طبیعی است و تیم می‌تواند بیشتر تست‌های رگرسیونی را با هم خودکارسازی کند. جادوی همکاری تیمی موفق آغاز می‌شود.

**به‌گونه‌ای کدنویسی کنید که گویی باید تا آخر عمر از آن پشتیبانی کنید**

*یوری زوبارف*

شما می‌توانید از 97 نفر بپرسید که هر برنامه‌نویسی باید چه بداند و انجام دهد، و ممکن است 97 پاسخ متفاوت دریافت کنید. این می‌تواند همزمان سردرگم‌کننده و ترسناک باشد. همه نصایح خوب است، همه اصول صحیح است، و همه داستان‌ها جذابند، اما از کجا شروع کنید؟ مهم‌تر از آن، وقتی شروع کردید، چگونه با تمامی بهترین شیوه‌هایی که یاد گرفته‌اید همگام شوید و آن‌ها را به بخشی جدایی‌ناپذیر از عملکرد برنامه‌نویسی خود تبدیل کنید؟

فکر می‌کنم پاسخ در حالت ذهن یا به طور ساده‌تر، در نگرش شما نهفته است. اگر به سایر توسعه‌دهندگان، تسترها، مدیران، بازاریابان، فروشندگان و کاربران نهایی اهمیت ندهید، سپس انگیزه‌ای برای به‌کارگیری توسعه‌ محور تست یا نوشتن توضیحات واضح در کد خود نخواهید داشت. فکر می‌کنم راه ساده‌ای برای تنظیم نگرش و همیشه انگیزه‌مند بودن برای تحویل محصولات باکیفیت‌تر وجود دارد:

*به‌گونه‌ای کد بنویسید که گویی باید تا آخر عمر از آن پشتیبانی کنید.*

همین. اگر این مفهوم را بپذیرید، اتفاقات شگفت‌انگیز بسیاری رخ خواهد داد. اگر بپذیرید هر یک از کارفرماهای گذشته یا فعلی شما حق دارند در میانه شب به شما زنگ بزنند و از شما بخواهند انتخاب‌هایی را که هنگام نوشتن متد fooBar انجام دادید توضیح دهید، به‌تدریج در مسیر تبدیل‌شدن به یک برنامه‌نویس حرفه‌ای پیشرفت خواهید کرد. شما به طور طبیعی می‌خواهید نام‌های بهتری برای متغیرها و متدها انتخاب کنید. از بلاک‌های کد شامل صدها خط دوری خواهید کرد. الگوهای طراحی را جستجو خواهید کرد، یاد خواهید گرفت و استفاده خواهید کرد. توضیحات خواهید نوشت، کد خود را تست خواهید کرد و به طور مداوم آن را رفکتور خواهید کرد. پشتیبانی از تمام کدهایی که تاکنون نوشته‌اید، برای بقیه عمرتان باید یک کار قابل مقیاس‌بندی باشد؛ بنابراین چاره‌ای جز بهبود، هوشمندتر شدن و کارآمدتر شدن نخواهید داشت.

اگر به آن فکر کنید، کدی که سال‌ها پیش نوشتید، چه بخواهید چه نخواهید، همچنان روی شغل شما تأثیر می‌گذارد. شما با هر روش، کلاس و ماژولی که طراحی و می‌نویسید دنباله‌ای از دانش، نگرش، سرسختی، حرفه‌ای بودن، سطح تعهد و میزان لذت خود را به جا می‌گذارید. مردم بر اساس کدی که می‌بینند درباره شما نظر می‌دهند. اگر آن نظرات دائماً منفی باشد، کمتر از آن چیزی که انتظار داشتید از شغل خود بهره خواهید برد. با هر خط کد مراقب شغل خود، مشتریان و کاربران خود باشید – طوری کد بنویسید که انگار باید تا آخر عمر از آن پشتیبانی کنید.

اگر به آن بیندیشید، کدی که سال‌ها پیش نوشتید هنوز حرفه شما را تحت‌تأثیر قرار می‌دهد، خواه آن را دوست داشته باشید یا نه. شما با هر متد، کلاس و ماژولی که طراحی می‌کنید و می‌نویسید، ردپایی از دانش، نگرش، پافشاری، حرفه‌ای‌گری، سطح تعهد و میزان لذت خود به جا می‌گذارید. مردم بر اساس کدی که می‌بینند، نظراتی در مورد شما خواهند داشت. اگر این نظرات مدام منفی باشند، کمتر از آنچه امیدوار بودید از حرفه خود به دست خواهید آورد. از حرفه، مشتریان و کاربران خود با هر خط کد مراقبت کنید - به‌گونه‌ای کدنویسی کنید که گویی باید تا آخر عمر از آن پشتیبانی کنید.

**با استفاده از مثال‌ها توابع کوچک بنویسید**

*کیت بریثویت*

می‌خواهیم کدی بنویسیم که درست باشد، و شواهدی در دست داشته باشیم که نشان دهد درست است. برای کمک به هر دو مورد، می‌تواند مفید باشد در مورد «اندازه» یک تابع فکر کنیم. نه در معنای مقدار کدی که یک تابع را پیاده‌سازی می‌کند - اگرچه آن جالب است - بلکه اندازه تابع ریاضی است که کد ما نمایان می‌کند.

به‌عنوان‌مثال، در بازی Go یک شرط به نام آتاری وجود دارد که در آن سنگ‌های یک بازیکن می‌توانند توسط حریف او زده شوند: یک سنگ با دو یا چند فضای آزاد مجاور به آن (که آزادی نامیده می‌شود) در آتاری نیست. ممکن است شمارش آزادی‌های یک سنگ دشوار باشد، اما تعیین آتاری اگر آن شناخته شده باشد، آسان است. ممکن است با نوشتن تابعی مانند این شروع کنیم:

boolean atari(int libertyCount)

libertyCount < 2

این بزرگ‌تر از آن چیزی است که به نظر می‌رسد. یک تابع ریاضی می‌تواند به‌عنوان یک مجموعه درک شود، زیرمجموعه‌ای از حاصل‌ضرب دکارتی مجموعه‌هایی که دامنه آن (اینجا، int) و محدوده آن (اینجا، bool) هستند. اگر این مجموعه‌های مقادیر همان اندازه‌ای بودند که در جاوا دارند، آنگاه 2^32\*(Integer.MAX\_VALUE+(−1∗Integer.MIN\_VALUE)+1) یا 8،589،934،592 عضو در مجموعه int\*bool وجود داشت. نیمی از این‌ها اعضای زیرمجموعه‌ای هستند که تابع ماست، بنابراین برای ارائه شواهد کامل از درستی تابعمان، نیاز داریم حدود 4.3 × 10^9 مثال را بررسی کنیم.

این اصل ادعایی است که تست‌ها نمی‌توانند عدم وجود باگ‌ها را ثابت کنند. اگرچه، تست‌ها می‌توانند نشان‌دهنده وجود ویژگی‌ها باشند، اما هنوز مسئله اندازه وجود دارد.

دامنه مسئله به ما کمک می‌کند. ماهیت Go به این معنی است که تعداد آزادی‌های یک سنگ هیچ int نیست، بلکه دقیقاً یکی از {1،2،3،4} است. بنابراین می توانیم به جای آن بنویسیم:

LibertyCount = {1,2,3,4}

boolean atari(LibertyCount libertyCount)

libertyCount == 1

این بسیار قابل‌مدیریت‌تر است: تابع محاسبه شده اکنون یک مجموعه با حداکثر هشت عضو است. در واقع، چهار نمونه بررسی شده شواهدی از اطمینان کامل ایجاد می‌کند که تابع درست است. این یکی از دلایلی است که استفاده از انواع ارتباط نزدیکی با قلمرو مسئله برای نوشتن برنامه‌ها، به‌جای انواع بومی، ایده خوبی است. استفاده از انواع الهام گرفته از قلمرو اغلب می‌تواند توابع ما را بسیار کوچک‌تر کند. یک راه برای دریافت اینکه این انواع چه باید باشند، پیداکردن نمونه‌هایی برای بررسی به عبارات قلمرو مسئله، قبل از نوشتن تابع است.

**برای بقیه تست بنویسید**

*جرارد مسزاروس*

شما در حال نوشتن تست‌های خودکار برای برخی یا همه کدهای تولیدی خود هستید. تبریک می‌گویم! شما تست‌ها را قبل از نوشتن کد می‌نویسید؟ حتی بهتر! فقط انجام این کار، شما را یکی از پذیرندگان اولیه در لبه پیشرفته عملکرد مهندسی نرم‌افزار می‌کند. اما آیا شما تست‌های خوب می‌نویسید؟ چگونه می‌توانید بگویید؟ یک راه این است که از خودتان بپرسید: «من این تست‌ها را برای چه کسی می‌نویسم؟» اگر پاسخ «برای خودم، برای ذخیره تلاش برای رفع باگ‌ها» یا «برای کامپایلر، بنابراین می‌توانند اجرا شوند» باشد، در این صورت احتمال دارد بهترین تست‌های ممکن را ننویسید. پس شما باید تست‌ها را برای چه کسی بنویسید؟ برای فردی که سعی در درک کد شما دارد.

تست‌های خوب به‌عنوان مستندات برای کدی که آن‌ها را تست می‌کنند، عمل می‌کنند. آن‌ها توصیف می‌کنند که کد چگونه کار می‌کند. برای هر سناریوی استفاده، تست (ها):

* کانتکست، نقطه شروع یا پیش‌نیازهایی را که باید برآورده شوند، توصیف می‌کنند.
* نشان می‌دهند که نرم‌افزار چگونه فراخوانی می‌شود.
* نتایج مورد انتظار یا پس‌شرط‌هایی را که باید تأیید شوند، توصیف می‌کنند.

سناریوهای مختلف استفاده، نسخه‌های کمی متفاوت از هر یک از این‌ها خواهند داشت. شخصی که سعی در درک کد شما دارد باید بتواند به چند تست نگاه کند و با مقایسه این سه بخش از تست‌های موردنظر، ببیند چه چیز باعث می‌شود نرم‌افزار رفتار متفاوتی داشته باشد. هر تست باید رابطه علت و معلولی بین این سه بخش را به‌وضوح نشان دهد.

این بدان معناست که آنچه در تست قابل‌مشاهده نیست، درست به اندازه آنچه قابل‌مشاهده است مهم است. کد زیادی در تست، خواننده را با جزئیات بی‌اهمیت حواس‌پرت می‌کند. هر جا امکان داشته باشد، چنین جزئیاتی را پشت فراخوانی‌های متد بامعنا پنهان کنید – ریفکتور کردن اکسترکت متد بهترین دوست شماست و مطمئن شوید به هر تست، نام معنی‌داری می‌دهید که سناریوی استفاده خاص را توصیف می‌کند تا خواننده تست مجبور نباشد هر تست را برای درک سناریوهای مختلف مهندسی معکوس کند. در مجموع، نام‌های کلاس تست و متد کلاس باید حداقل نقطه شروع و نحوه فراخوانی نرم‌افزار را شامل شوند. این امکان را فراهم می‌کند که پوشش تست از طریق اسکن سریع نام‌های متد تأیید شود. همچنین مفید است که نتایج مورد انتظار را در نام‌های متد تست ذکر کنید مشروط بر اینکه این کار باعث نشود نام‌ها خیلی بلند شوند که نتوان آن‌ها دید یا خواند.

این هم یک ایده خوب است که تست‌های خود را تست کنید. می‌توانید تأیید کنید که آن‌ها خطاهایی را که فکر می‌کنید تشخیص می‌دهند، با قراردادن این خطاها در کد تولید، تشخیص می‌دهند (البته نسخه خصوصی خودتان که دور خواهید انداخت). مطمئن شوید آن‌ها خطاها را به روشی مفید و بامعنا گزارش می‌دهند. شما همچنین باید تأیید کنید که تست‌های شما به طور واضح با فردی که سعی در درک کد شما دارد، صحبت می‌کنند. تنها راه انجام این کار این است که کسی که با کد شما آشنا نیست، تست‌های شما را بخواند و به شما بگوید چه چیزی یاد گرفته است. به‌دقت به آنچه می‌گوید، گوش کنید. اگر چیزی را به طور واضح درک نکرد، احتمالاً به این دلیل نیست که او بسیار باهوش نیست. احتمال بیشتر این است که شما بسیار روشن نبودید. (جلو بروید و نقش‌ها را با خواندن تست‌های او معکوس کنید!)

**شما باید نسبت به کد اهمیت قائل شوید**

*پیت گودلیف*

کار شرلوک هولمز نیست که بفهمد برنامه‌نویسان خوب، کد خوب می‌نویسند. برنامه‌نویسان بد... نمی‌نویسند. آن‌ها هیولاهایی تولید می‌کنند که بقیه ما باید آن‌ها را تمیز کنیم. شما می‌خواهید چیزهای خوب بنویسید، درست است؟ شما می‌خواهید یک برنامه‌نویس خوب باشید.

کد خوب از هوا ظاهر نمی‌شود. چیزی نیست که اتفاقی زمانی که سیارات در یک ردیف قرار می‌گیرند، رخ دهد. برای دست‌یافتن به کد خوب، باید روی آن کار کنید، سخت و تنها زمانی کد خوب خواهید داشت که واقعاً به کد خوب اهمیت دهید.

برنامه‌نویسی خوب فقط از صلاحیت فنی زاده نمی‌شود. من برنامه‌نویسان بسیار باهوشی دیده‌ام که می‌توانند الگوریتم‌های شدید و تحسین‌برانگیز تولید کنند، کسانی که استاندارد زبان خود را حفظ هستند، اما بدترین کدها را می‌نویسند. خواندن، استفاده و اصلاح آن دردناک است. من برنامه‌نویسان فروتن‌تری دیده‌ام که به کد بسیار ساده چسبیده‌اند، اما برنامه‌های ظریف و بیانی می‌نویسند که لذت‌بخش است با آن‌ها کار کرد.

بر اساس سال‌ها تجربه من در کارخانه نرم‌افزار، نتیجه گرفته‌ام که اختلاف واقعی بین برنامه‌نویسان متوسط و برنامه‌نویسان عالی این است: نگرش. برنامه‌نویسی خوب در اتخاذ رویکرد حرفه‌ای و میل به نوشتن بهترین نرم‌افزاری که می‌توانید، در محدودیت‌ها و فشارهای دنیای واقعی کارخانه نرم‌افزار، نهفته است.

جاده‌ای به‌سوی جهنم با نیت‌های نیک فرش شده است. برای اینکه یک برنامه‌نویس عالی باشید، باید از نیت خوب فراتر روید و واقعاً به کد اهمیت دهید - دیدگاه‌های مثبت را پرورش دهید و نگرش‌های سالم ایجاد کنید. کد عالی بادقت توسط استادکاران ساخته می‌شود، نه اینکه بی‌دقت توسط برنامه‌نویسان سهل‌انگار هک شده باشد و یا به طور مرموزی توسط خودخوانده‌های ادعایی کدنویسی ساخته شده باشد.

شما می‌خواهید کد خوب بنویسید. شما می‌خواهید یک برنامه‌نویس خوب باشید. پس، به کد اهمیت می‌دهید:

* در هر وضعیت کدنویسی، از هک‌کردن چیزی که فقط به نظر می‌رسد کار می‌کند، خودداری می‌کنید. تلاش می‌کنید کد ظریفی بنویسید که به‌وضوح درست باشد (و تست‌های خوبی داشته باشد که نشان دهد درست است).
* کدی می‌نویسید که قابل کشف باشد (برنامه‌نویسان دیگر بتوانند آن را به‌راحتی برداشته و درک کنند)، قابل نگه‌داشتن باشد (که شما، یا برنامه‌نویسان دیگر، بتوانید در آینده به‌راحتی آن را اصلاح کنید) و درست باشد (همه گام‌های ممکن برای تعیین اینکه مسئله را حل کرده‌اید، نه فقط به نظر می‌رسد برنامه کار می‌کند را انجام می‌دهید).
* شما به‌خوبی در کنار سایر برنامه‌نویسان کار می‌کنید. هیچ برنامه‌نویسی جزیره‌ای مجزا نیست. تعداد کمی از برنامه‌نویسان به‌تنهایی کار می‌کنند. اکثر آنها در یک تیم از برنامه‌نویسان، یا در محیط شرکت و یا در یک پروژه اوپن سورس مشغول به کار هستند. شما برنامه‌نویسان دیگر را در نظر می‌گیرید و کدی می‌نویسید که دیگران بتوانند بخوانند. شما می‌خواهید تیم بهترین نرم‌افزار ممکن را بنویسد، نه اینکه خودتان را باهوش نشان دهید.
* هر بار که بخشی از کد را لمس می‌کنید، تلاش می‌کنید آن را بهتر از زمانی که پیدایش کردید، ترک کنید (ساختار بهتری داشته باشد، بهتر تست شده باشد، قابل‌فهم‌تر باشد...).
* شما به کد و برنامه‌نویسی اهمیت می‌دهید، بنابراین به طور مداوم زبان‌ها، اصطلاحات و تکنیک‌های جدید یاد می‌گیرید. اما آن‌ها را فقط زمانی که مناسب باشد، اعمال می‌کنید.

خوشبختانه، شما این مجموعه نصایح را می‌خوانید؛ چون به کد اهمیت می‌دهید. شما را سر ذوق می‌آورد. این شوروشوق شماست. از برنامه‌نویسی لذت ببرید. از برش کد برای حل مسائل حیله‌گرانه، لذت ببرید. نرم‌افزاری تولید کنید که باعث افتخار شما شود.

**مشتریان شما منظورشان آن چیزی نیست که می‌گویند**

*نیت جکسون*

من هنوز مشتری‌ای ندیده‌ام که خیلی خوشحال نباشد تا به من بگوید چه می‌خواهد - معمولاً با جزئیات زیاد. مشکل این است که مشتریان همیشه کل حقیقت را نمی‌گویند. آنها به‌طورکلی دروغ نمی‌گویند، اما به زبان مشتری صحبت می‌کنند، نه به زبان توسعه‌دهنده. از اصطلاحات و بافت‌های خودشان استفاده می‌کنند. جزئیات مهمی را حذف می‌کنند. فرض‌هایی می‌کنند که شما 20 سال است در شرکت آنها بوده‌اید، دقیقاً مثل خودشان. این موضوع با این واقعیت تشدید می‌شود که بسیاری از مشتریان در واقع نمی‌دانند چه می‌خواهند! برخی ممکن است درکی از «تصویر کلی» داشته باشند، اما به ندرت قادرند جزئیات چشم‌انداز خود را به طور مؤثر ارتباط دهند. دیگران ممکن است در چشم‌انداز کامل سبک‌تر باشند، اما می‌دانند چه چیز نمی‌خواهند. پس چگونه می‌توانید پروژه نرم‌افزاری را به کسی تحویل دهید که به شما کل حقیقت در مورد آنچه می‌خواهد را نمی‌گوید؟ کاملاً ساده است. فقط بیشتر با آنها تعامل داشته باشید.

مشتریان خود را زود و به طور مکرر به چالش بکشید. صرفاً آنچه را که به شما گفتند می‌خواهند، با کلمات خودشان تکرار نکنید. به خاطر داشته باشید: منظورشان چیزی نبود که به شما گفتند. من اغلب این نصیحت را با جایگزین‌کردن کلمات مشتری در گفتگو با آن‌ها و قضاوت واکنش آن‌ها اجرا می‌کنم. شگفت‌زده خواهید شد که اصطلاح مشتری چقدر معنای متفاوتی نسبت به اصطلاح کلاینت دارد. بااین‌حال، شخصی که به شما می‌گوید در پروژه نرم‌افزاری خود چه می‌خواهد، از این اصطلاحات به طور مترادف استفاده می‌کند و انتظار دارد شما هم متوجه شوید که او در مورد کدام یک صحبت می‌کند. شما گیج خواهید شد و نرم‌افزاری که می‌نویسید آسیب خواهد دید.

موضوعات را قبل از اینکه تصمیم بگیرید که نیاز مشتری را درک کرده‌اید، چندین بار با مشتریان خود بحث کنید. سعی کنید مسئله را دو یا سه بار برای آن‌ها بازگو کنید. در مورد چیزهایی که قبل یا بعد از موضوع موردبحث اتفاق می‌افتد، با آن‌ها صحبت کنید تا درک بهتری داشته باشید. اگر امکانش هست، افراد مختلف را وادار کنید در گفتگوهای جداگانه در مورد یک موضوع به شما بگویند. آنها تقریباً همیشه برای شما داستان‌های متفاوتی تعریف خواهند کرد که واقعیت‌های جدا اما مرتبطی را آشکار می‌کند. دو نفر که در مورد یک موضوع صحبت می‌کنند، اغلب با یکدیگر تناقض خواهند داشت. بهترین فرصت موفقیت شما این است که قبل از شروع ساخت نرم‌افزار فوق پیچیده خود، تفاوت‌ها را حل‌وفصل کنید.

از کمک‌های دیداری در گفتگوهای خود استفاده کنید. این می‌تواند به‌سادگی استفاده از یک وایت‌برد در جلسه باشد، به‌راحتی ایجاد یک ماکت دیداری اولیه در مرحله طراحی باشد یا به پیچیدگی ساخت یک پروتوتایپ عملکردی باشد. به‌طورکلی شناخته شده است که استفاده از کمک‌های دیداری در طول یک گفتگو به طول انداختن دوره توجه ما کمک می‌کند و نرخ بازیابی اطلاعات را افزایش می‌دهد. از این واقعیت استفاده کنید و پروژه خود را برای موفقیت آماده کنید.

در زندگی گذشته، من یک «برنامه‌نویس چندرسانه‌ای» در یک تیم بودم که پروژه‌های درخشان تولید می‌کرد. یکی از مشتریان ما افکار خود را در مورد ظاهر و احساس پروژه را با جزئیات زیاد شرح داد. طرح رنگی کلی که در جلسات طراحی موردبحث قرار گرفت، پس‌زمینه سیاه را برای ارائه نشان داد. فکر کردیم آن را میخکوب کرده‌ایم. تیم‌های طراحان گرافیکی شروع به تولید صدها فایل گرافیکی لایه‌بندی شده کردند. زمان زیادی صرف شکل‌دادن به محصول نهایی شد. در روزی که محصول ثمره زحماتمان را به مشتری نشان دادیم، خبرهای شگفت‌انگیزی دریافت کردیم. وقتی او محصول را دید، دقیقاً در مورد رنگ پس‌زمینه گفت: «وقتی گفتم سیاه، منظورم سفید بود.» پس، می‌بینید، هیچ‌گاه به‌وضوح سیاه‌وسفید نیست.

1. the Pragmatic Programmers [↑](#footnote-ref-1)
2. Don’t Repeat Yourself [↑](#footnote-ref-2)
3. Open/Closed Principle [↑](#footnote-ref-3)
4. Single Responsibility [↑](#footnote-ref-4)
5. Plain Old Java Object [↑](#footnote-ref-5)
6. Plain Old C# Object or Plain Old CLR Object [↑](#footnote-ref-6)
7. You Aren’t Gonna Need It [↑](#footnote-ref-7)
8. <http://www.processing.org> [↑](#footnote-ref-8)
9. The C Programming Language [↑](#footnote-ref-9)
10. <http://www.amazon.com/dp/0135974445/> [↑](#footnote-ref-10)
11. \* <http://www.developerdotstar.com/mag/articles/reeves_design.html> [↑](#footnote-ref-11)
12. Design by Contract [↑](#footnote-ref-12)
13. \* <http://www.netjeff.com/humor/item.cgi?file=ApolloComputer> [↑](#footnote-ref-13)
14. Framework for Integrated Test [↑](#footnote-ref-14)
15. acceptance test–driven development [↑](#footnote-ref-15)