97 چیز که هر برنامه‌نویسی باید بداند

ویرایش شده توسط کِولین هِنی

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

لینکدین:
https://www.linkedin.com/in/sasan-safari-45a374222

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 قطعه-

ای 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 عذاب 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

پِچیل 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] 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.

[1] the Pragmatic Programmers

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 بِرِگ 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 تجزیه-کننده) 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 کردن چیزها نترسید

مایک لوئیس

هرکسی که تجربه کار 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 این است در یک آسمان‌خراش با تیرهای I شکل بازی جنگا انجام دهیم که 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 کد 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 این 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 تبدیل کنید و حین این کار تست‌های مختلف انجام دهید. تلاش برای انجام یک بازسازی بزرگ در یک مرحله، مشکلات زیادی ایجاد می‌کند که ممکن است شما را مجبور می‌کند کل پروژه را در میانه راه رها کنید.

مثل جراحي باشید که از بریدن قسمت‌هاي بدن بیمار براي عمل هراس نداشته باشد. این نگرش مسری است و دیگران را ترغیب می‌کند تا روی پروژه‌هایی که نیاز به پاکسازی دارند و به تعویق انداخته‌اند، کار کنند. فهرستی از وظایف «بهداشتي» داشته باشید 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

دیروقت 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 شركت‌ها از عناوین آهنگ‌های Sex Pistols استفاده كردم. براي نماد سهام فقط چند كلمه چهارحرفی با حروف بزرگ نياز داشتم.

از كلمات ركيك چهارحرفی استفاده كردم.

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 دید.

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 پیغام 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 قاتی 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، دیالوگ 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 نادیده نگیرید!

پیت گودلیف

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

پایم درد گرفت، اما عجله داشتم تا دوستانم را ببینم؛ بنابراین، پس بلند شدم و به راهم ادامه دادم. هرچه بیشتر 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 دیگر 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.

بسیاري از برنامه‌نویسان مثل شب بد من برنامه می‌نویسند.

خطا، چه خطایی؟ جدی نیست، 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 ذخیره 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 مدیریت خطاها پشتیبانی می‌شوند. شما نمی‌توانید به‌راحتی آن‌ها را نادیده بگیرید. البته می‌شود! من کدهاي این چنیني زیادي 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 مشکلات رسیدگي کنید.

عدم رسیدگی به خطاها منجر به این مشکلات می‌شود:

- کد شکننده. کدهايي که پر از باگ‌های هیجان‌انگیز هستند و سخت پیدا 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 اینترفیس‌هاي 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 ددلاین من هم نزدیك است.
- می‌دانم که این فراخوانی تابع هرگز خطایی را برنخواهد گرداند (printf همیشه کار می‌کند، malloc همیشه حافظه جدید را برمی‌گرداند—اگر این اتفاق نیفتد، 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.