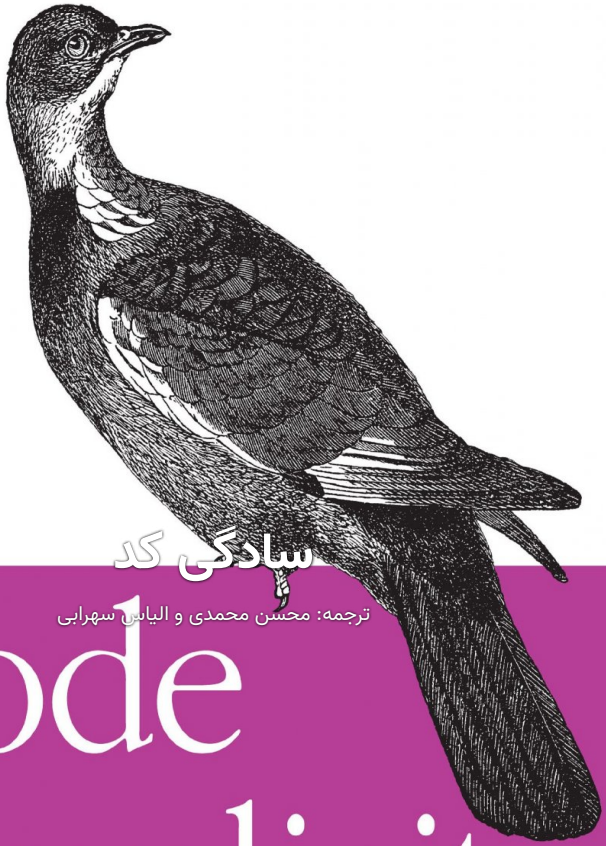
The Fundamentals of Software



سادگی کد

ترجمه: محسن محمدی و الیاس سهرابی

Code Simplicity

O'REILLY®

Max Kanat-Alexander

پیش گفتار

این کتاب تلاش می‌کند تا ایده‌هایی درباره روش درست طراحی نرم‌افزار، با ارائه قوانین و قواعد به شما بدهد. همچنین این کتاب می‌کوشد تا طراحی و توسعه نرم‌افزار را همراه با مفاهیم فلسفی که اغلب درک آن ساده است، ارائه دهد. بعضی از مطالب و ایده ها در کتاب شناخته شده هستند اما بعضی دیگر تازگی دارند. مزیت این کتاب انسجام تمامی مطالب آن است.

شما میتوانید با تمرکز و به کارگیری این مطالب قادر به استفاده بهینه از زمان خود باشید. همچنین این ایده ها برای توسعه دهندگان مبتدی بسیار مورد نیاز است و باید هر روز تمرین و استفاده شود. این کتاب به صورت رایگان (noneprofit) ترجمه شده است. برای ارسال نظرات و مشکلات احتمالی در متن کتاب با ایمیل های زیر ارتباط برقرار کنید.

mohsenm@pm.me

mh.sohrabi1578@gmail.com

فهرست کتاب

فصل 1

مقدمه

تفاوت بین یک برنامه نویس بد و یک برنامه‌نویس خوب, درک کردن است. به همین دلیل است که برنامه نویسان بد کاری که انجام می‌دهند را درک نمی‌کنند ولی برنامه نویسان خوب درک می کنند. باور کنید یا نه، واقعا همین قدر ساده است.

این کتاب برای کمک به همه برنامه نویسان برای درک توسعه نرم‌افزار به کار می‌رود. در سطح گسترده ای که می تواند برای هر زبان برنامه نویسی یا مقاله ای استفاده شود

اگر شما یک برنامه نویس هستید ، این قوانین به شما توضیح می دهد که چرا برخی از روش های توسعه نرم افزار کار می کنند و برخی از آنها کار نمی کنند. آنها به شما در تصمیم گیری روزمره درباره توسعه نرم افزار کمک می کنند همچنین به تیم شما کمک می کنند مکالمه های هوشمندانه ای داشته باشند که در آخر منجر به برنامه های منطقی شود.

اگر شما یک برنامه‌نویس نیستید اما در صنعت نرم‌افزار کار می‌کنید، می‌توانید این کتاب را برای خود به چند دلیل مفید بدانید:

- این کتاب یک ابزار آموزشی عالی برای استفاده در آموزش برنامه نویسان تازه کار است، در حالی که حاوی اطلاعاتی است که به شدت به برنامه نویسان ارشد مربوط است.
- این کتاب به شما این امکان را می‌دهد که به طور موثر درک کنید که چرا مهندسین نرم‌افزار می‌خواهند برخی کارهای خاص را انجام دهند، یا اینکه چرا نرم‌افزار باید به روش خاصی توسعه یابد.
- همچنین می‌تواند به شما کمک کند ایده‌های خود را به طور موثر برای مهندسان نرم‌افزار توضیح دهید ، به شما کمک می کند اصول اساسی را که مهندسان نرم افزار خوب تصمیمات خود را بر اساس آنها میگیرند ، درک کنید.

در حالت ایده آل ، هر کسی که در صنعت نرم افزار کار می کند باید قادر به خواندن و درک این کتاب باشد, حتی اگر تجربه زیادی در برنامه‌نویسی نداشته باشد، یا اگر انگلیسی زبان مادری آن‌ها نباشد. داشتن درک فنی بیشتر به شما در فهم برخی مفاهیم کمک میکند, اما بیشتر آنها به هیچ تجربه ای نیاز ندارند.

در واقع، با وجود اینکه این کتاب در مورد توسعه نرم‌افزار است، تقریبا هیچ کد نوشته شده ای ندارد. چطور چنین چیزی ممکن است؟ خوب، ایده این است که این اصول بتوانند در هر پروژه نرم‌افزاری، در هر زبان برنامه‌نویسی، اعمال شوند.

شما لازم نیست یک زبان برنامه‌نویسی مشخص را برای درک چیزهایی که همه جا در برنامه‌نویسی کاربرد دارند، بدانید.

در عوض، مثال‌های دنیای واقعی و قیاس در سراسر کتاب مورد استفاده قرار می‌گیرند تا به شما کمک کنند درک بهتری از هر اصل داشته باشید، همانطور که ارائه شده‌است.

مهم تر از همه، این کتاب برای کمک به شما و همچنین برای کمک به افزودن سلامت، نظم و سادگی به زمینه توسعه نرم‌افزار نوشته شده‌است. امیدوارم از خواندن آن لذت ببرید و به نوعی زندگی و نرم افزار شما را بهبود ببخشد.

تقدیر و تشکر

ویراستاران من، Andy Oram و Jolie Kanat منبعی بسیار ارزشمند بوده‌اند. بازخورد اندی هم خردمندانه و هم درخشان بود. اصرار و حمایت جولی در نهایت باعث شد که این کتاب منتشر شود، و کارهای ویرایشی گسترده وی در پیش نویس های اولیه بسیار مورد استقبال قرار گرفت.

مدیر تحریریه من ، راشل هد ، استعداد قابل توجهی در حل کردن و بهبود همه چیز دارد.

همه برنامه نویسانی که با آنها در جامعه متن باز کار کرده ام و با آنها صحبت کرده ام نیز شایسته تشکر هستند به ویژه از سایر توسعه دهندگان من در پروژه Bugzilla که به من کمک کردند تمام ایده های این کتاب را در یک سیستم نرم افزاری واقعی و زنده طی سالیان زیاد امتحان کنم.

نظرات و بازخوردی که من در طول این سال‌ها در وبلاگ خود دریافت کرده‌ام، به من کمک کرده‌است تا شکل و محتوای این کتاب را شکل دهم. هر کسی که در آنجا مشارکت کرده، مستحق تشکر است، حتی آن‌هایی که به سادگی مرا تشویق کردند یا به من اطلاع می دادند که یک مقاله من را خوانده اند.

در سطح شخصی بسیار ممنونم از جون میلان ، کتی ویور ، و همه کسانی که با آنها کار می کنند. به معنای واقعی ، آنها باعث این هستند که الان من می توانم این کتاب را بنویسم. و سرانجام ، کلاه من به دوستم رون و در آخر، از دوست عزیزم رون بسیار تشکر میکنم. بدون او نوشتن این کتاب اصلا امکان نداشت.

فصل 2

معرفی

کامپیوتر ها یک تغییر عمده در جامعه ایجاد کرده اند. دلیل این است که آن‌ها ما را قادر می‌سازند که کار بیشتری با افراد کمتر انجام دهیم. این ارزش یک کامپیوتر است - می‌تواند خیلی سریع کار کند. فوق العادست. اما مشکل این است که رایانه‌ها خراب میشوند. آن‌ها بطور مداوم خراب میشوند. اگر چیز دیگری در خانه شما به اندازه کامپیوتر شما خراب میشد، آن را باز می‌گرداندید. اکثر مردم در جوامع مدرن حداقل یک‌بار در روز با یک کامپیوتر که خراب میشود یا به درستی عمل نمیکند برخورد میکنند. این اصلا خوب نیست.

چه چیزی در مورد کامپیوترها اشتباه است؟

چرا کامپیوترها اینقدر خراب می شوند؟ برای نرم افزار ، تنها یک دلیل وجود دارد . فقط برنامه نویسی بد. برخی افراد مدیریت را مقصر می دانند و برخی دیگر مشتریان را ، اما تحقیقات نشان می دهد که ریشه اصلی این مشکل همیشه برنامه نویسی است.

اما منظور ما از "برنامه نویسی بد" چیست؟ این یک اصطلاح بسیار مبهم است. و برنامه نویسان به طور کلی افراد بسیار باهوشی و منطقی هستند - چرا برخی از آنها بد برنامه نویسی میکنند؟

اساسا، همه چیز حول پیچیدگی می‌گردد. کامپیوتر احتمالاً پیچیده ترین وسیله ای است که امروزه می توانیم در کارخانه تولید کنیم. در هر ثانیه میلیاردها محاسبه انجام می دهد. صدها میلیون از قطعات کوچک الکترونیکی وجود دارد که همه باید به درستی عمل کنند.

برنامه ای که روی کامپیوتر نوشته می شود به همان اندازه پیچیده است. به عنوان مثال ، هنگام نوشتن، Microsoft Windows 2000 یکی از بزرگترین برنامه هایی بود که تاکنون ایجاد شده است ، تقریباً در حدود 30 میلیون خط کد نوشته شد. نوشتن این مقدار کد ، چیزی شبیه نوشتن یک کتاب 200.000.000 کلمه ای است - بیش از پنج برابر دانشنامه بریتانیکا.

پیچیدگی یک برنامه می تواند به طور خاصی گیج کننده باشد, زیرا هیچ چیزی وجود ندارد که به آن دسترسی داشته باشید. وقتی از کار بیفتد, نمی توانید چیزی قابل اطمینان پیدا کنید که بتوان آن را بررسی کرد. سروکار داشتن با چنین موضوعی می تواند بسیار سخت باشد, زیرا کاملا انتزاعی است؛ در واقع یک برنامه ساده کامپیوتری به قدری پیچیده است که هیچکس نمیتواند درک کند که تمام کد ها چگونه به طور کامل کار میکنند. هرچه برنامه ها گسترده تر می شوند , این موضوع بیشتر نمایان میشود.

بنابراین, برنامه نویسی باید عملی باشد برای تبدیل پیچیدگی به سادگی. در غیر این صورت , وقتی یک برنامه به سطح بالایی از پیچیدگی رسید , کسی نمیتواند به کار کردن با آن ادامه دهد. قسمت های پیچیده یک برنامه باید به شکلی ساده ساماندهی شوند تا یک برنامه نویس بتواند بدون داشتن توانایی های دور از انتظار بر روی آنها کار کند.

این همان هنر و استعدادی است که در برنامه نویسی دخیل است "تبدیل پیچیدگی به سادگی"

یک "برنامه نویس بد" شخصی است که قادر به کاهش پیچیدگی نیست. این یک اشتباه رایج است. زیرا مردم باور دارند با نوشتن کدی که "فقط کار میکند" در حال کاهش پیچیدگی نوشتار زبان برنامه نویسی (که به خودی خود بسیار پیچیده است)

هستند. کمی شبیه مثال زیر است :

مهندسی را تصور کنید که به وسیله ای نیاز دارد تا یک میخ را با آن به داخل زمین بکوبد سپس با قرقره , نخ و یک آهنربای بزرگ دستگاهی را می سازد . احتمالاً فکر می کنید خیلی مسخره باشد.

حالا تصور کنید کسی به شما می گوید , "من به کدی نیاز دارم که بتوانم در هر برنامه ای و در هر مکانی از آن استفاده کنم، که میتواند بین دو کامپیوتر ارتباط برقرار کند." قطعاً تبدیل آن به چیزی ساده دشوار است . بنابراین ، برخی از برنامه نویسان(شاید بیشتر برنامه نویسان) در آن شرایط راه حلی ارائه می دهند که مشابه مثال نخ و قرقره و یک آهنربای بزرگ است که به سختی قابل فهم برای افراد دیگر است. آنها غیر منطقی نیستند و هیچ مشکلی ندارند. آنها وقتی با یک کار واقعاً دشوار روبرو می شوند ، درزمان کوتاهی که دارند آنچه را که می توانند انجام می دهند. آنچه آن ها می سازند تا جایی که به آنها مربوط باشد کارساز خواهد بود. کاری که قرار است انجام دهد را انجام میدهد. این چیزی است که رئیس آن ها و همچنین مشتریانشان میخواهند.

اما به هر حال ، آنها نتوانسته اند پیچیدگی را به سادگی تبدیل کنند. سپس آنها این برنامه را به یک برنامه نویس دیگر منتقل می کنند ، و آن برنامه نویس با استفاده از آن به عنوان بخشی از برنامه اش، به پیچیدگی آن اضافه میکند. هرچقدر افراد بیشتری نسبت به کاهش پیچیدگی بی توجهی کنند, به همان نسبت برنامه نامفهوم تر میشود.

با نزدیک شدن یک برنامه به پیچیدگی بی پایان ، پیدا کردن همه مشکلات آن غیرممکن می شود. هزینه هواپیماهای جت میلیون ها و میلیاردها دلار است چرا که آنها دارای چنین پیچیدگی هستند و "اشکال زدایی" شده اند. اما بیشتر نرم افزار ها حدودا $50 الی $100 برای مشتری هزینه دارند. با این قیمت ، هیچ کس وقت یا منابع لازم را نخواهد داشت که همه مشکلات را از یک سیستم بی نهایت پیچیده برطرف کنید.

به راستی برنامه کامپیوتری چیست؟

عبارت "یک برنامه کامپیوتری" ، به شکلی که اکثر مردم از آن استفاده می کنند ، دو تعریف کاملاً مشخص دارد:

۱. دنباله ای از دستورالعمل های داده شده به کامپیوتر.

۲. اعمال انجام‌شده توسط کامپیوتر در نتیجه دستورات داده شده.

معنی اولین تعریف همان چیزی است که برنامه نویسان در هنگام نوشتن یک برنامه می‌بینند. تعریف دوم همان چیزی است که کاربران زمانی می‌بینند که از یک برنامه استفاده می‌کنند. برنامه‌نویس به رایانه می‌گوید: " یک خوک را روی صفحه نمایش بده ". این تعریف ۱ است.کامپیوتر مقدار زیادی الکتریسیته را جابجا میکند که باعث می‌شود یک خوک بر روی صفحه ظاهر شود. و این تعریف ۲ است، اقدامات انجام‌شده توسط کامپیوتر. هر دو, برنامه‌نویس و کاربر در اینجا خواهند گفت که با "یک برنامه کامپیوتری" کار می‌کنند، اما تجربه هر یک آن‌ها از آن برنامه بسیار متفاوت است. برنامه نویس ها با کلمات و نمادها کار می‌کنند، در حالی که کاربران تنها نتیجه نهایی را می‌بینند.(اقدامات صورت‌گرفته)

در نهایت یک برنامه کامپیوتری هر دو این کارها را انجام می‌دهد: دستورالعمل‌هایی که برنامه‌نویس آنها را می‌نویسد و کارهایی که کامپیوتر انجام می‌دهد. هدف کلی از نوشتن این دستورالعمل این است که باید کاری انجام شود - بدون انجام این کارها، هیچ دلیلی برای نوشتن دستورالعمل وجود ندارد. درست مثل زندگی است، مثل وقتی یک لیست خرید را می‌نویسید. این یک دستورالعمل برای خرید در فروشگاه است.اگر فقط دستورها را می‌نوشتید، اما هیچ وقت به فروشگاه نمیرفتید، این کار بی‌فایده بود. دستورالعمل ها باید باعث شوند که اتفاقی بیفتد.

تفاوت قابل توجهی بین نوشتن یک لیست خرید برای مواد غذایی و نوشتن یک برنامه کامپیوتری وجود دارد. اگر لیست مواد غذایی شما نامرتب باشد ،این امر فقط کمی زمان خرید شما را کند می کند. اما اگر کد برنامه شما بی نظم باشد ، دستیابی به اهداف شما می تواند به یک پیچ و خم دشوار تبدیل شود. چرا این طور هست؟ خوب ، لیست های مواد غذایی کوتاه و ساده هستند ، و وقتی کار با آنها تمام شد آنها را دور میریزید. برنامه‌های کامپیوتری بزرگ و پیچیده هستند و شما اغلب باید آن‌ها را برای سال‌ها یا دهه‌ها حفظ کنید. بنابراین ، در حالی که فقط یک لیست مواد غذایی ساده می تواند شما را دچار مشکل کند ، هر چقدر هم که بی نظم باشد اما کد سازماندهی نشده کامپیوتر می تواند واقعاً مشکل زیادی برای شما ایجاد کند.

علاوه بر این، هیچ زمینه دیگری وجود ندارد که در آن مجموعه دستورالعمل ها و نتیجه این دستورالعمل ها به اندازه ای که در زمینه توسعه نرم افزار هست ، به هم پیوند خورده باشند.

در زمینه‌های دیگر، مردم دستورالعمل ها را می‌نویسند و سپس آن‌ها را به افراد دیگر تحویل می‌دهند و اغلب منتظر میمانند تا اجرای آنها را شاهد باشند. برای مثال، هنگامی که یک معمار ساختمان یک خانه را طراحی می‌کند، مجموعه‌ای از دستورالعمل‌ها را می‌نویسد. این دستورات باید در طی یک زمان طولانی از افراد مختلف عبور کنند تا به یک ساختمان فیزیکی تبدیل شوند. ساختمان نهایی نتیجه برداشت و نظرات مردم از دستورالعمل معمار است. از سوی دیگر، وقتی کد می‌نویسیم، کسی بین ما و کامپیوتر وجود ندارد. نتیجه دقیقا همان چیزی است که دستورالعمل برای انجام آن گفته شده است.(بدون هیچ سوالی)

کیفیت نتیجه نهایی به طور کامل وابسته به کیفیت ماشین، کیفیت ایده‌های ما، و کیفیت کد ما است.

از بین این سه عامل، کیفیت کد بزرگ‌ترین مشکلی است که امروزه پروژه‌های نرم‌افزاری با آن روبرو میشوند. به همین دلیل و سایر نکاتی که در بالا ذکر شد، بیشتر این کتاب در مورد بهبود کیفیت کد است. ما بر روی ایده‌ها و ماشین‌ها و همچنین در چند مکان تمرکز می‌کنیم، اما بیشتر تمرکز بر بهبود ساختار و کیفیت کد است که به دستگاه می‌دهیم. اگرچه ما زمان زیادی را صرف صحبت در مورد کد می‌کنیم، اگرچه وقت زیادی را صرف صحبت در مورد کد می کنیم ، بسیار آسان است فراموش کنیم که این کار را صرفاً انجام می دهیم زیرا ما خواهان نتیجه بهتر هستیم.

هیچ چیز در این کتاب نتیجه ضعیف را نمی بخشد ، دلیل اصلی این که ما بر بهبود دادن کد تمرکز داریم این است که بهبود دادن کد مهمترین مشکلی است که باید برای بهبود دادن نتیجه انجام دهیم.

فصل 3

علم گمشده

این کتاب بیشتر در مورد چیزی به نام "طراحی نرم‌افزار" است، شاید شما این عبارت را قبلا شنیده باشید و شاید حتی برخی از کتاب‌های مربوط به آن را خوانده باشید. اما اجازه دهید با تعاریف دقیق و جدید به آن نگاه کنیم.

ما می دانیم که معنی کلمه "نرم افزار" چیست. بنابراین آنچه باید معنی کنیم کلمه "طراحی" است:

فعل:

1. برنامه ریزی برای آفرینش. مثال: مهندس یک پل را در این ماه طراحی خواهد کرد و در ماه آینده آن را خواهد ساخت.

اسم:

1. نقشه ای که برای ساختن چیزی به وجود آمده است که تا کنون ساخته نشده. مثال: مهندس نقشه ای را برای یک پل ارائه داده. او ماه آینده آن را خواهد ساخت.
2. نقشه ای که یک چیز بوجود آمده از آن پیروی می کند. مثال: آن پل در آنجا از طراحی خوبی را پیروی می کند.

• تمامی این تعاریف زمانی اعمال می‌شوند که ما در مورد طراحی نرم‌افزار صحبت می‌کنیم:

زمانی که ما در حال طراحی نرم‌افزار ("طراحی" به عنوان یک فعل) هستیم، در حال برنامه‌ریزی آن هستیم.

چیزهای زیادی برای برنامه ریزی در مورد نرم‌افزار وجود دارد - ساختار کد، فن‌آوری‌های مورد استفاده و غیره. تصمیمات فنی زیادی برای گرفتن وجود دارد. اغلب، ما فقط این تصمیمات را در ذهن خود می‌گیریم، اما گاهی هم برنامه‌های خود را یادداشت و یا نمودارهایی را برای آن رسم می‌کنیم.

• زمانی که این کار را انجام دادیم، نتیجه یک "طراحی نرم‌افزاری" است ("طراحی" به عنوان اولین تعریف اسم). این طرحی است که ما ساخته‌ایم. یک‌بار دیگر، این ممکن است یک سند نوشتاری باشد، اما همچنین می‌تواند یک مشت تصمیماتی باشد که ما در ذهن خود نگه می‌داریم.

کدی که در حال حاضر وجود دارد، "طرحی" (طرح " به عنوان تعریف دوم اسم) دارد، که ساختاری است که دارد یا طرحی است که باز آن پیروی می‌کند. برخی کدها ممکن است فاقد ساختار واضحی باشند - "کد هیچ طرحی ندارد، " به این معنی که هیچ برنامه مشخصی توسط برنامه نویس اصلی ایجاد نشده است. بین "بدون طراحی" و "یک طراحی" درجات مختلفی وجود دارد همچون طراحی جزیی , "چندینطرحمتناقضدریکقطعهکد" "یک طراحی کامل" و غیره. همچنین طرح های بدی وجود دارند که از "بدون طرح" بودن بدتر هستند. به عنوان مثال، تصور کنید که با یک کد که به طور عمدی باعث ایجاد بی‌نظمی یا پیچیدگی شده باشد روبرو شده اید. این یک کد با طراحی بد فعال است .

علم طراحی نرم‌افزار یک علم برای تصمیم‌گیری و برنامه ریزی در مورد نرم‌افزار است.این به مردم کمک می‌کند که تصمیماتی اینچنین بگیرند:

- ساختار کد برنامه ما چگونه باید باشد؟
- آیا تمرکز بر داشتن برنامه سریع یا برنامه‌ای که خواندن کد آن آسان است، اهمیت بیشتری دارد؟
- برای نیازهای خود از کدام زبان برنامه نویسی استفاده کنیم؟

طراحی نرم افزار موارد زیر را شامل نمی شود:

- ساختار شرکت چه خواهد بود؟
- چه زمانی باید جلسات تیمی داشته باشیم؟
- برنامه نویسان چه زمانی باید کار کنند؟
- چگونه باید عملکرد برنامه نویسان خود را اندازه گیری کنیم؟

اینها تصمیماتی درباره نرم افزار شما نیستند، بلکه تصمیماتی درباره شما یا سازمان شما هستند. قطعا انجام این تصمیمات مهم است - بسیاری از پروژه‌های نرم‌افزاری شکست‌خورده اند زیرا مدیریت ضعیف دارند. اما این هدف کتاب نیست. این کتاب در مورد نحوه تصمیم گیری صحیح فنی درباره نرم افزار است.

هر چیزی که شامل معماری سیستم نرم‌افزاری شما باشد و یا تصمیمات فنی که شما در هنگام ایجاد سیستم میگیرید تحت عنوان " طراحی نرم‌افزار " قرار می‌گیرد.

هر برنامه نویس یک طراح است

هر برنامه نویس که روی یک پروژه نرم افزاری کار می کند درگیر طراحی است. توسعه دهنده اصلی مسئول طراحی معماری کل برنامه است. برنامه نویسان ارشد مسئولیت طراحی مناطق بزرگ مربوط به خود را بر عهده دارند. و برنامه نویسان تازه کار طراحی قطعات خود در برنامه را بر عهده دارند ، حتی اگر مسئولیتشان به سادگی تغییر دادن یک قسمت از یک فایل باشد. حتی در نوشتن یک خط کد مقدار مشخصی از طراحی دخیل است.

حتی وقتی همه چیز را خودتان برنامه نویسی میکنید ، بازهم پروسه طراحی ادامه دارد. بعضی اوقات بلافاصله قبل از ضربه انگشتانتان به صفحه کلید تصمیم می گیرید و این کل مرحله طراحی است. و گاهی اوقات شب در رختخواب به این فکر میکنید که میخواهید چطور برنامه را بنویسد.

هر کسی که نرم‌افزار می‌نویسد یک طراح است. هر فرد در یک تیم نرم‌افزاری مسئول حاصل اطمینان از این است که کد خود آن‌ها به خوبی طراحی شده‌است. هر کسی که برای یک پروژه نرم افزاری کد می نویسد ، نمی تواند طراحی نرم افزار را در هر سطحی که باشد نادیده بگیرد.

اگرچه، این بدان معنا نیست که طراحی یک دموکراسی است. شما نباید بر اساس نظر گروهی طراحی کنید - نتیجه یک طراحی فعال بد خواهد بود. در عوض ، تمام توسعه دهندگان باید این اختیار را داشته باشند که در زمینه طراحی های خود تصمیمات خوبی بگیرند. اگر آنها تصمیمات ضعیف یا متوسط بگیرند ، باید توسط یک توسعه دهنده ارشد یا برنامه نویس اصلی نادیده گرفته شود ، کسانی که باید حق وتو بر طراحان زیردستشان داشته باشد. اما در غیر این صورت مسئولیت طراحی کد باید با افرادی باشد که واقعا روی آن کار می‌کنند. یک طراح همیشه باید مایل به گوش دادن به پیشنهادات و بازخورد باشد زیرا برنامه نویسان معمولاً افراد باهوشی هستند که ایده های خوبی دارند. اما بعد از بررسی تمام داده‌ها هر تصمیمی باید توسط یک فرد و نه توسط گروهی از افراد گرفته شود.

علم طراحی نرم‌افزار

طراحی نرم افزار ، همانطور که امروزه در دنیا آموزش داده میشود، یک علم نیست. علم چیست؟ تعریف فرهنگ لغت کمی پیچیده است، ، اما اساساً برای اینکه یک موضوع علمی باشد ، باید آزمون های خاصی را پشت سر بگذارد:

- یک علم باید حاوی دانش جمع آوری شده باشد. یعنی باید از واقعیت ها تشکیل شود - نه از نظرات - و این حقایق باید جایی جمع شده باشند (مانند کتاب).
- این دانش باید نوعی سازماندهی داشته باشد. باید آن را در دسته بندی کرد ، قطعات مختلف از نظر اهمیت باید به درستی با یکدیگر مرتبط باشند و غیره.
- علم باید حاوی حقایق کلی یا قوانین اساسی باشد.
- علم باید به شما بگوید که در دنیای فیزیکی چگونه کار کنید. در حقیقت باید به گونه‌ای در کار و یا در زندگی کاربرد داشته باشد.
- معمولا، یک علم کشف می‌شود و از طریق روش علمی به اثبات می‌رسد، که شامل مشاهده جهان فیزیکی، انجام آزمایش‌ها برای تایید نظریه شما، و نشان دادن این که همان آزمایش همه جا کار می‌کند تا نشان دهد که نظریه یک واقعیت عمومی است و نه صرفا یک تصادف یا چیزی که فقط برای شما کار می‌کند

در دنیای نرم افزار ، دانش زیادی داریم که در کتابها جمع آوری شده است ، و حتی تا حدودی سازمان یافته است. با این حال ، از بین تمام موارد لازم برای ساختن یک علم ، مهمترین قسمت را از دست داده ایم: قوانینی که به وضوح بیان شده اند و حقایق غیر قابل تغییری که هرگز ما را ناامید نمیکنند.

توسعه دهندگان باتجربه نرم افزار می دانند کار صحیح چیست ، اما چرا این کار صحیح است؟ چه چیزی باعث می شود برخی تصمیمات درست و برخی تصمیمات اشتباه باشد؟

قوانین اساسی طراحی نرم‌افزار کدامند؟

این کتاب دارای سابقه کشف آنها است. مجموعه ای از تعاریف ، حقایق ، قواعد و قوانین را برای توسعه نرم افزار بیان می کند، که بیشتر روی طراحی نرم افزار متمرکز است. تفاوت بین یک واقعیت ، یک قاعده ، یک تعریف و یک قانون چیست؟

• تعاریف به شما می گویند که چیزی چیست و چگونه از آن استفاده کنید.

• حقایق فقط گفته های درست درباره چیزی هستند. هر اطلاعات واقعی یک واقعیت است.

• قواعد گزاره هایی هستند که به شما مشاوره واقعی می دهند، موارد خاصی را پوشش می دهند و به شما کمک می کنند.

تصمیمات را در بهترین راستا بگیرید ، اما لزوماً به شما در پیش بینی آنچه در آینده اتفاق می افتد کمک نمی کند یا اینکه واقعیت های دیگر را کشف کنید آن‌ها معمولا به شما می‌گویند که اقدام کنید یا نه.

• قوانین حقایقی هستند که همیشه درست خواهند بود و حوزه وسیعی از دانش را در بر می گیرند.

آنها به شما کمک می کنند حقایق مهم دیگر را رقم بزنید و به شما امکان می دهد آنچه در آینده اتفاق خواهد افتاد را پیش بینی کنید.

از بین همه اینها ، قوانین مهمترین هستند. در این کتاب ، شما خواهید دانست که چه چیزی قانون است زیرا متن صریحاً چنین گفته است.اگر مطمئن نیستید که برخی از اطلاعات در کدام گروه قرار می گیرند، پیوست دوم تمام اطلاعات اصلی کتاب

را لیست می کند و آنها را به وضوح به عنوان یک قانون، یک قاعده و یک تعریف یا یک حقیقت کاملاً قدیمی برچسب گذاری می کند.

وقتی برخی از این تعاریف ، قوانین یا حقایق را می خوانید، ممکن است با خود بگویید: "این واضح بود ، من قبلاً این را می دانستم." در واقع کاملاً انتظار می رفت - اگر مدت ها در دنیای توسعه نرم افزار بوده اید، احتمالاً قبلاً با برخی از این ایده ها روبرو شده اید. با این حال، هنگامی که این واکنش را دارید، از خود بپرسید:

- آیا می‌دانستم که این قطعه خاص از داده‌ها به طور مطمئنی صحیح است؟
- آیا می‌دانستم چقدر مهم بود؟
- آیا می‌توانستم آن را به طور واضح به شخص دیگری بگویم تا به طور کامل آن را درک کند؟
- آیا من نحوه ارتباط آن با داده‌های دیگر در زمینه توسعه نرم‌افزار را درک کردم؟

اگر می‌توانید به برخی از این پرسش‌ها "بله" بگویید در حالی که قبلا گفته بودید "نه" یا "شاید"، شما یک نوع درک خاص به دست آوردید، و این درک بخش عظیمی از چیزی است که دانش را از مجموعه‌ای از ایده‌ها متمایز می‌سازد.

البته ممکن است این علم هنوز کامل نباشد. همیشه چیزی بیشتر برای کشف در جهان وجود دارد ، در هر زمینه ای نیز مطالب زیادی برای دانستن وجود دارد. گاهی اوقات حتی اصلاحاتی در قوانین پایه وجود دارد که باعث میشود داده‌های جدید کشف یا حقایق جدید بوجود آید.اما این نقطه شروع است! این چیزی است که می‌تواند براساس یک مجموعه واقعی از قوانین و حقایق قابل‌مشاهده برای ایجاد نرم‌افزار ساخته شود.

حتی اگر بخش‌هایی از این کتاب روزی اشتباه ثابت شوند و علمی بهتر توسعه یابد، مهم این است که یک واقعیت روشن باشد: *طراحی نرم‌افزار می‌تواند یک علم باشد*. این یک راز همیشگی نیست، به نظر هر برنامه‌نویس یا هر مشاوری که می‌خواهد چند دلار را برای فروش "روش جدید" خود برای طراحی نرم‌افزار بدست آورد:

قوانین وجود دارند، آن‌ها را می توان تشخیص داد و شما می‌توانید آن‌ها را بشناسید.آن‌ها ابدی، بی‌تغییر و اساسا درست هستند و کار می‌کنند.

در مورد اینکه آیا قوانین این کتاب قوانین صحیحی هستند یا نه ... خوب ، صدها مثال و آزمایش وجود دارد که می تواند برای اثبات آنها ذکر شود ، اما در پایان ، شما باید خودتان تصمیم بگیرید. قوانین را آزمایش کنید. ببینید آیا می توانید به واقعیتهای کلی در مورد توسعه نرم افزار که گسترده تر یا اساسی تر فکر کنید. و اگر به هر موردی رسیدید یا مشکلی در قوانین پیدا کردید ، برای نحوه تماس با نویسنده در مورد مشارکتها یا سولات خود به www.codesimplicity.comمراجعه کنید.هر تحول بعدی در این موضوع به نفع همه خواهد بود، به شرطی که واقعاً این پیشرفت ها ، قوانین اساسی یا قوانین طراحی نرم افزار باشد.

چرا هیچ علمی در زمینه طراحی نرم افزار وجود نداشته است؟

شاید برای شما جالب باشد که بدانید چرا قبل از این کتاب، علمی برای طراحی نرم افزار وجود نداشته است. به هر حال ، ما دهه ها است که نرم افزار می نویسیم.

خوب ، این یک داستان جالب است. در اینجا برخی از پیش زمینه ها آورده شده است که ممکن است به شما کمک کند بدون توسعه دانش طراحی نرم افزار، چطور ما تاکنون با کامپیوترها کنار آمده ایم.

آنچه امروزه به عنوان "کامپیوتر" در ذهن داریم ، در ذهن ریاضیدانان به عنوان دستگاههای کاملاً انتزاعی آغاز شده است - افکاری در مورد چگونگی حل مسائل ریاضی با استفاده از ماشین آلات به جای ذهن. این ریاضیدانان افرادی هستند که ما آنها

را بنیانگذاران علوم کامپیوتر(یعنی مطالعه ریاضی پردازش اطلاعات) می دانیم.

اما برخلاف باور مردم که این کار، مطالعه برنامه‌نویسی کامپیوتر نیست. با این حال ، اولین کامپیوترها تحت نظارت این دانشمندان کامپیوتر توسط مهندسان الکترونیک بسیار ماهر ساخته شده اند. آنها توسط اپراتورهای بسیار آموزش دیده در محیط های کاملاً کنترل شده اداره می شدند. همه آنها توسط سازمانهایی که به آنها نیاز داشتند (بیشتر دولتها برای هدف گیری موشکها و شکستن کد ها) ساخته شده اند و تنها یک یا دو نسخه از هر مدل ارایه‌شده وجود داشت.

سپس UNIVAC و اولین رایانه های تجاری جهان به بازار آمدند. در این مرحله، تعداد محدودی ریاضیدان نظری پیشرفته در جهان وجود داشت. وقتی کامپیوترها در دسترس همه قرار گرفتند ، به طور واضح امکان ارسال ریاضیدان به همراه هرکدام وجود نداشت. بنابراین اگرچه برخی از سازمان ها، مانند اداره سرشماری ایالات متحده (یکی از اولین دریافت کنندگان UNIVAC) ،مطمئناً اپراتورهای بسیار آموزش دیده ای برای ماشین آلات خود داشتند، اما سازمان های دیگر بدون شک دستگاه های خود را تهیه کردند و گفتند: "خوب آقای بیل از حسابداری، این مال تو است! کتابچه راهنما را بخوان و قبوض را در آن قرار بده! " و بیل رفت ، در این ماشین پیچیده شروع به کار کرد و تمام تلاش خود را کرد تا کار کند.

بدین ترتیب بیل یکی از اولین "برنامه نویسان فعال" شد. او ممکن است ریاضیات را در مدرسه خوانده باشد ، اما مطمئناً نظریات پیشرفته در مورد نیازها برای طراحی ماشین را مطالعه نکرده است. اما هنوز هم می توانست کتابچه راهنما را بخواند و آن را درک کند و با آزمایش و خطا ، دستگاه را وادار کند آنچه را که می خواست انجام دهد.

البته ، هرچه رایانه های تجاری بیشتری ارسال می شدند ،بیشتر به بیل و اپراتورهای آموزش دیده کمتری نیاز داشتیم. اکثر قریب به اتفاق برنامه نویسان دقیقاً مانند بیل هستند. مدیریت از او تقاضاهایی داشت که "اکنون آن کار را انجام بده!" و به او گفته شد، "برای ما مهم نیست که چگونه این کار انجام می شود، فقط آن را انجام بده!" او فهمید که چگونه کار را طبق کتابچه راهنمای خود انجام دهد. حتی اگر هر دو ساعت یک بار خراب میشد.

بالاخره بیل یک گروه از برنامه نویسان را به کار گرفت تا با او کار کنند. او باید بفهمد که چگونه یک سیستم را طراحی کند و کارها را بین افراد مختلف تقسیم کند. در کل هنر برنامه نویسی عملی, بصورت ارگانیک رشد کرد ، آنها بیشتر شبیه دانشجویان دانشگاهی هستند که به خودشان آشپزی می آموزند تا مهندسان ناسا که شاتل فضایی را می ساختند.

بنابراین در این مرحله، یک سیستم شلوغ و به هم ریخته از توسعه نرم‌افزار وجود دارد، و مدیریت آن بسیار پیچیده و دشوار است، اما همه به طریقی موفق می‌شوند. سپس به همراه "*The Mythical Man Month*" (انتشارات ادیسون - وسلی)، کتابی به قلم فرد بروکس که در واقع به فرآیند توسعه نرم‌افزار در یک پروژه واقعی نگاه کرد و به برخی واقعیات را در مورد آن اشاره کرد - بسیار معروف است که اضافه کردن برنامه نویسان به یک پروژه نرم‌افزاری دیرهنگام آن را بعدا می‌سازد. او با یک علم کامل آشنا نشد، اما مشاهدات خوبی در مورد برنامه‌نویسی و مدیریت توسعه نرم‌افزار انجام داد.

بعد از آن، مجموعه‌ای از روش‌های توسعه نرم‌افزار پدیدار شدند: فرآیند اتحاد منطقی، مدل بلوغ قابلیت، توسعه نرم‌افزار چابک، و بسیاری دیگر. هیچ یک از اینها ادعا نمی‌کردند که یک علم باشند - آن‌ها فقط راه‌هایی برای مدیریت پیچیدگی توسعه نرم‌افزار بودند.

و این اساسا ما را به جایی می‌رساند که امروزه هستیم: روش‌های زیادی، اما بدون هیچ دانش واقعی. در واقع، دو علوم از دست رفته وجود دارد: علم مدیریت نرم‌افزار و علم طراحی نرم‌افزار.

دانش مدیریت نرم افزار به ما می گوید که چگونه کار را بین برنامه نویسان تقسیم کنیم ، چگونه زمان انتشار را برنامه ریزی کنیم ، چطور می توان زمان کار را تخمین زد و غیره.

هم اکنون به طور فعال روی این دانش کار می شود و با روش های مختلف ذکر شده در بالا به آن پرداخته می شود. این واقعیت که به نظر می رسد نظرات متناقض و به همان اندازه معتبر در این زمینه وجود دارد، نشان می دهد که هنوز قوانین اساسی مدیریت نرم افزار تدوین نشده است. با این حال ، حداقل به مسئله توجه شده است.

از طرف دیگر ، علم طراحی نرم افزار در دنیای عملی برنامه نویسی چندان مورد توجه قرار نمی گیرد. تعداد کمی از افراد در مدرسه آموخته اند که علمی برای طراحی نرم افزار می تواند وجود داشته باشد. در عوض ، بیشتر به آنها گفته می شود ، "این زبان برنامه نویسی به این ترتیب کار می کند. حالا برو چند نرم افزار بنویس! "

این کتاب برای پر کردن این خلاء نوشته شده است.

علمی که در اینجا ارائه می شود علوم کامپیوتر نیست. علوم کامپیوتر مطالعه ریاضی است. در عوض ، این کتاب شامل آغازی علمی برای "برنامه نویسان شاغل" است - مجموعه ای از قوانین و قواعد اساسی که هنگام نوشتن یک برنامه به هر زبانی باید رعایت شود. این علمی است که به اندازه فیزیک یا شیمی قابل اعتماد است درباره اینکه چطور یک برنامه بسازید.

گفته شده است که چنین علمی ممکن نیست زیرا طراحی نرم افزار بسیار متغیر است و هرگز با قوانین ساده و اساسی توصیف نمی شود. و همه اینها فقط نظریات است. بعضی از افراد نیز زمانی گفتند که درک جهان فیزیکی غیرممکن است زیرا "این آفرینش خداست و خدا غیر قابل شناخت است" ، و با این وجود ما دانش هایی را برای جهان فیزیکی کشف کردیم. بنابراین، مگر اینکه شما اعتقاد داشته باشید که رایانه ها غیر قابل شناخت هستند، ساختن علم طراحی نرم افزار کاملاً ممکن است.

همچنین یک افسانه رایج وجود دارد که برنامه نویسی کاملاً نوعی هنر است و کاملاً مطابق با خواسته های شخصی هر برنامه نویس است. با این حال، گرچه درست است که هنرهای مختلفی در استفاده از هر علمی دخیل هستند، اما هنوز هم باید علمی برای استفاده وجود داشته باشد ، و در حال حاضر هیچ علمی برای نرم افزار وجود ندارد. در واقع مشکل اصلی پیچیدگی در نرم افزار ، فقدان این علم است. اگر برنامه نویسان واقعاً از نظر علمی، سادگی در نرم افزار داشته باشند، تقریباً اینقدر پیچیدگی وجود نخواهد داشت و برای مدیریت این پیچیدگی به فرآیندهای دیوانه وار احتیاج نداریم.

فصل 4

نیرو های محرک طراحی نرم افزار

زمانی که نرم‌افزاری می‌نویسیم، باید ایده‌ای از اینکه چرا این کار را انجام می‌دهیم و هدف نهایی چیست، داشته باشیم.

آیا راهی وجود دارد که بتوانیم هدف همه نرم‌افزارها را جمع بندی کنیم؟ اگر چنین بیانیه‌ای امکان پذیر باشد، به کل علم طراحی نرم‌افزار جهت می‌دهد، زیرا ما می‌دانیم درحال انجام چه چیزی هستیم.

خوب، در حقیقت یک هدف واحد برای همه نرم‌افزارها وجود دارد:

برای کمک به مردم

ما می توانیم این مساله را به یک هدف خاص برای هر بخش از نرم افزار تقسیم کنیم. به عنوان مثال ، یک پردازشگر کلمه برای کمک به مردم در نوشتن چیزهایی وجود دارد و یک مرورگر وب برای کمک به افراد در مرور وب وجود دارد.

برخی از نرم افزارها فقط برای کمک به گروه های خاصی از افراد وجود دارند. به عنوان مثال ، بسیاری از نرم افزارهای حسابداری وجود دارد که برای کمک به حسابداران وجود دارد. اینها فقط آن گروه خاص از مردم را هدف قرار می دهند.

در مورد نرم افزاری که به حیوانات یا گیاهان کمک می کند چطور؟ خوب، هدف آن واقعاً کمک به مردم برای کمک به حیوانات یا گیاهان است.

نکته مهم در اینجا این است که نرم افزاری هرگز برای کمک به اشیا بی جان وجود ندارد. نرم افزار برای کمک به رایانه وجود ندارد، بلکه همیشه برای کمک به مردم وجود دارد. حتی وقتی کتابخانه می نویسید، برای کمک به برنامه نویسان که مردم هستند، می نویسید. شما هرگز برای کمک به رایانه چیزی نمی نویسید.

حال، "کمک" به چه معناست؟ از بعضی جهات، ذهنی است - آنچه به یک شخص کمک می کند ممکن است به دیگری کمک نکند. اما این کلمه تعریف فرهنگ لغت دارد ، بنابراین:

معنای کلمه به طور کامل در اختیار هر فرد نیست. فرهنگ لغت زبان جدید جهان آمریکایی Webster "کمک" را چنین تعریف می کند:

> آسان تر کردن کار (یک شخص) مشخصاً ... برای انجام بخشی از کار؛ سهولت کار یا تقسیم کار

موارد زیادی وجود دارد که می توانید به مردم کمک کنید - تنظیم برنامه، نوشتن کتاب، برنامه ریزی رژیم غذایی و هر چیز دیگری. آنچه به شما کمک می کند بستگی به خود شما دارد اما هدف همیشه کمک است.

هدف نرم افزار "درآمدزایی" یا "نشان دادن میزان هوشمند بودن" نیست. هرکسی که تنها با این اهداف برنامه نویسی کند, هدف نرم افزار را نقض کرده و احتمالاً به دردسر می افتد. مسلماً ، این روشها میتواند به شما "کمک" کند ، اما این محدوده کمک بسیار محدود است و طراحی فقط با این اهداف ممکن است منجر به تولید نرم افزاری با کیفیت پایین تر نسبت به طراحی واقعی که برای کمک به افراد و انجام کاری که نیاز دارند و یا می‌خواهند انجام دهند، شود.

افرادی که نمی توانند کمک به شخص دیگری را تصور کنند، نرم افزار بدی می نویسند - یعنی نرم افزار آنها کمک چندانی به مردم نخواهد کرد. در حقیقت ، ممکن است این نظریه (به عنوان حدس ، براساس مشاهدات بسیاری از برنامه نویسان در طول زمان) وجود داشته باشد که توانایی بالقوه شما در نوشتن یک نرم افزار خوب فقط به توانایی تصور کمک به دیگری محدود می شود.

به طور کلی، زمانی که در حال تصمیم‌گیری درباره نرم‌افزار هستیم، اصل راهنمای ما می‌تواند این باشد که چگونه می‌توانیم کمک کنیم. (و به یاد داشته باشید، درجات مختلفی از کمک وجود دارد که فرد می‌تواند به تعداد زیادی یا چند نفر کمک کند) شما حتی می‌توانید ویژگی ها را اولویت‌بندی کنید. کدام ویژگی بیشترین کمک افراد کمک می‌کند؟ و آن ویژگی باید در اولویت قرار گیرد. مطالب بیشتری در مورد اولویت‌بندی ویژگی‌ها وجود دارد، اما "این کار چقدر به کاربران ما کمک می‌کند؟" سوال اساسی خوبی است که درباره هر گونه تغییر پیشنهادی در سیستم نرم‌افزاری باید پرسید.

به طور کلی ، این هدف - کمک به مردم - مهمترین چیزی است که باید هنگام طراحی نرم افزار به خاطر بسپارید، و اکنون تعریف آن به ما این امکان را می دهد که یک علم واقعی در زمینه طراحی نرم افزار ایجاد و درک کنیم.

برنامه دنیای واقعی

چگونه می توانیم هدف نرم افزار را در پروژه های خود در دنیای واقعی اعمال کنیم؟ خوب ، بگذارید بگوییم ما در حال نوشتن یک ویرایشگر متن برای برنامه نویسان هستیم. اولین کاری که باید انجام دهیم تعیین هدف نرم افزار است. بهتر است ساده باشد، بنابراین بگذارید بگوییم هدف "کمک به برنامه نویسان برای ویرایش متن" است. خوب است که خصوصیات بیشتری داشته باشد، و گاهی اوقات مفید است، اما اگر گروه نتواند در مورد هدف خاصی به توافق برسد، حداقل یک هدف ساده مانند هدف بالا را ارائه دهید.

اکنون که هدف داریم، بیایید تمام ویژگی های درخواستی را بررسی کنیم. برای هر یک از این ویژگی ها ما می توانیم از خود بپرسیم، "این ویژگی چگونه به برنامه نویسان کمک می کند که متن را بهتر ویرایش کنند؟" اگر پاسخ این باشد: "نمی شود"، می توانیم بلافاصله آن ویژگی را از لیست خود خارج کنیم. سپس، برای هر یک از ویژگی های باقی مانده، می توانیم جواب را به عنوان یک جمله کوتاه یادداشت کنیم. به عنوان مثال، فرض کنید شخصی از ما میخواهد میانبرهای صفحه کلید را برای اقدامات معمول اضافه کنیم. می توانیم بگوییم، "این ویژگی به برنامه نویسان کمک می کند متن را ویرایش کنند زیرا به آنها امکان می دهد بدون وقفه طولانی در تایپ با برنامه تعامل سریعتری داشته باشند." (در صورتی که برای شرایط شما عملی به نظر نمی رسد، شما مجبور نیستید این موارد را یادداشت کنید فقط داشتن ایده برای پاسخ خودتان کافی است.)

همچنین چند دلیل مفید دیگر برای پرسیدن این سوال وجود دارد:

- این کار به حل ابهامات در مورد توصیف این ویژگی یا نحوه پیاده‌سازی آن کمک می‌کند. به عنوان مثال، پاسخ بالا در مورد میانبرهای صفحه‌کلید به ما می‌گوید که پیاده‌سازی باید سریع باشد، چون کاربران ارزشی از آن خارج می‌شوند.
- این به تیم کمک می‌کند تا درباره ارزش یک ویژگی به توافق برسند. برخی افراد ممکن است ایده میان‌بر صفحه‌کلید را دوست نداشته باشند، اما همه باید بتوانند توافق کنند که پاسخ در بالا توضیح می‌دهد که چرا آن‌ها ارزشمند هستند. در حقیقت، برخی از توسعه دهندگان حتی ممکن است ایده بهتری از نحوه اجرای نیاز کاربر (تعامل با ویرایشگر متن)بدون میان‌بر صفحه‌کلید داشته باشند. خوبه! اگر پاسخ ما را به یک ایده بهتر از ویژگی باشد، باید آن را به جای آن اجرا کنیم. پاسخ به ما می‌گوید چیزی که واقعا مورد نیاز است، تنها چیزی که کاربر تصور می‌کند او می‌خواهد نیست.

- پاسخ به این پرسش آشکار می‌کند که برخی از ویژگی‌ها مهم‌تر از بقیه هستند. این کار به مدیران پروژه کمک می‌کند تا کار خود را اولویت‌بندی کنند.
- در بدترین حالت، اگر ویرایشگر متن ما با ویژگی‌های بسیار زیادی در طول زمان متورم شده باشد، پاسخ می‌تواند به ما کمک کند تصمیم بگیریم کدام ویژگی‌ها باید حذف شوند.

ما همچنین می توانیم لیستی از اشکالات ایجاد کنیم، که می توانیم آنها را مرور کنیم و سوال مخالف را بپرسیم: "این اشکال چگونه مانع ویرایش متن برنامه نویسان می شود؟" بعضی اوقات پاسخ واضح است ، بنابراین در واقع نیازی به نوشتن نیست. به عنوان مثال ، اگر هنگام ذخیره فایل، برنامه خراب می شود، نیازی به توضیح دلیل بد بودن آن نیست.

به احتمال زیاد روشهای بی شماری دیگری برای کاربردی کردن هدف یک نرم افزار در کارهای روزمره وجود دارد. اینها فقط چند نمونه هستند.

اهداف طراحی نرم‌افزار

اکنون که هدف این نرم افزار را دانستیم، می توانیم کمی به علم طراحی نرم افزار خود جهت دهیم. از این هدف، ما می‌دانیم که وقتی نرم‌افزاری را می‌نویسیم، می‌خواهیم به مردم کمک کنیم. بنابراین، یکی از اهداف علم طراحی نرم‌افزار باید این باشد:

اجازه دادن به ما برای نوشتن نرم‌افزاری که تا حد امکان مفید است.

دوم، ما معمولا از مردم می‌خواهیم که از کمک نرم افزار ما استفاده کنند. بنابراین هدف دوم ما این است:

برای اینکه به نرم افزار ما کمک کند تا حد ممکن مفید باشد.

اکنون، این یک هدف عالی است، اما هر سیستم نرم افزاری در هر اندازه ای بسیار پیچیده است ، بنابراین کار بسیار سختی است که با گذشت زمان آن را مفید نگه داریم. در حقیقت، مشکل اصلی نوشتن و نگهداری نرم افزار مفید, امروزه طراحی و برنامه نویسی است. وقتی ایجاد یا اصلاح نرم افزار دشوار باشد، برنامه نویسان بیشتر وقت خود را صرف تمرکز بر "درست کار کردن" عملیات ها می کنند و زمان کمتری را نیز صرف کمک به کاربر می کنند.اما وقتی کار روی یک سیستم آسان است، برنامه نویسان می توانند زمان بیشتری را صرف تمرکز بر مفید بودن برای کاربر و زمان کمتری را روی جزئیات برنامه نویسی اختصاص دهند. به همین ترتیب، نگهداری از یک نرم افزار راحت تر و اطمینان از مفید بودن نرم افزار برای برنامه نویسان آسان تر است.

این ما را به هدف سوم می رساند:

برای طراحی سیستم هایی که بتوانند به آسانی توسط برنامه نویسان ایجاد و نگهداری شوند، باید تا بتوانند - و تا آنجا که ممکن است - برای کاربران مفید باشند.

این هدف سوم همان هدفی است که به عنوان هدف طراحی نرم افزار تصور می شود، حتی اگر هرگز به صراحت بیان نشده باشد. با این حال، داشتن اهداف اول و دوم برای راهنمایی بسیار مهم است. ما می خواهیم به یاد داشته باشیم که مفید بودن در حال حاضر و در آینده انگیزه های هدف سوم است.

نکته ای که در مورد این هدف سوم باید به آن اشاره شود عبارت "به آسانترین شکل ممکن" است. ایده این است که ایجاد و نگهداری از برنامه های ما آسان باشد، نه اینکه آنها را دشوار یا پیچیده کند. این بدان معنا نیست که همه چیز بلافاصله آسان خواهد شد - گاهی اوقات برای یادگیری یک فناوری جدید و یا طراحی مطلوب زمان لازم است - اما در طولانی مدت، انتخاب های شما, ایجاد و نگهداری از نرم افزار شما را آسان تر می کند.

گاهی اوقات هدف اول (مفید بودن) و هدف سوم (سهولت در نگهداری) کمی در تضاد هستند - مفید بودن نرم افزار شما می تواند نگهداری و حفظ آن را دشوار کند. با این حال ، این دو هدف در طول تاریخ بیش از آنچه لازم بوده در تعارض بوده اند. ایجاد یک سیستم کاملاً قابل نگهداری که برای کاربران آن بسیار مفید باشد کاملاً امکان پذیر است. در حقیقت ، اگر آن را قابل نگهداری نکنید ، تحقق هدف دوم یعنی مفید بودن بسیار دشوار است. بنابراین ، هدف سوم مهم است زیرا در غیر این صورت دو هدف دیگر محقق نمی شوند.

فصل 5

آینده

اولین سوالی که طراحان نرم افزار با آن روبرو هستند این است: "چگونه می توانم در مورد نرم افزارم تصمیم بگیرم؟" وقتی با بسیاری از مسیرهای ممکن مواجه شدید، کدام گزینه بهترین است؟ هرگز این سوال وجود ندارد که کدام تصمیم کاملاً درست است در مقابل کدام تصمیم کاملاً اشتباه است. در عوض ، آنچه می خواهیم بدانیم این است: "با توجه به بسیاری از تصمیمات احتمالی ، کدام یک از این تصمیمات بهتر از دیگران است؟" این مسئله رتبه بندی تصمیمات و سپس انتخاب بهترین تصمیم از بین همه امکانات است.

به عنوان مثال، یک طراح می‌تواند از خود بپرسید " ۱۰۰ ویژگی متفاوت وجود دارد که ما امروزه می‌توانیم روی آن کار کنیم، اما ما نیروی انسانی داریم که فقط میتواند روی ۲ ویژگی کار کند. کدام یک را باید اول شروع کنیم؟ "

معادله طراحی نرم افزار

سوال فوق و در واقع هر سوال از این ماهیت در طراحی نرم افزار ، با این معادله پاسخ داده می شود:

$$D = \frac{V}{E}$$

بر این اساس

D

این نشانه میزان تمایل به تغییر است. چقدر تمایل به انجام کاری داریم؟

V

ارزش یک تغییر است. این تغییر چقدر ارزشمند است؟ معمولاً این را با پرسیدن این که "چقدر به کاربران ما کمک می کند؟" تعیین می کنید. اگرچه روشهای دیگری نیز برای تعیین ارزش وجود دارد.

E

نشان دهنده تلاشی است که در اجرای یک تغییر بکار میرود. این تغییر نیازمند چه مقدار تلاش است است؟

اساساً، این معادله می گوید:

مطلوبیت هر تغییر مستقیماً با ارزش تغییر متناسب است و بالعکس با تلاشی که در ایجاد تغییر انجام می شود متناسب است.

مشخص نیست که آیا تغییر کاملا درست است یا غلط؛ به جای آن، به شما می‌گوید که چگونه گزینه‌های خود را رتبه بندی کنید. تغییراتی که ارزش زیادی به همراه خواهند داشت و نیازمند تلاش کمی هستند "بهتر" از آن‌هایی هستند که ارزش کمی دارند و نیازمند تلاش زیادی هستند.

حتی اگر سوال تان این است که آیا باید همین طور بمانیم و تغییر نکنیم؟ این معادله به شما پاسخ می‌دهد. از خودتان بپرسید: "ثابت ماندن چه ارزشی دارد؟" و "تلاش لازم برای ثابت ماندن چیست؟" و آن را با ارزش تغییر و تلاش برای تغییر مقایسه کنید.

ارزش

منظور ما از "ارزش" در این معادله چیست؟ ساده‌ترین تعریفی که از ارزش ارائه می‌شود این است:

جایی که این تغییر به هر کسی در هر جایی کمک می‌کند.

مهم‌ترین افرادی که به شما کمک می‌کنند کاربران شما هستند. با این حال، نوشتن با ویژگی هایی که شما را از نظر مالی حمایت کند نیز نوعی ارزش است - یا به نوعی برای شما ارزشمند است. در حقیقت، روش های مختلفی وجود دارد که یک تغییر می تواند ارزش داشته باشد. اینها فقط دو نمونه است.

بعضی اوقات، تعیین مقدار عددی دقیق هر تغییر خاص دشوار است. به عنوان مثال ، بگویید نرم افزار شما به کاهش وزن افراد کمک می کند. چگونه مقدار دقیق کمکتان به کاهش وزن در کسی را اندازه می گیرید؟ واقعاً نمی توانید. اما می توانید با دقت بدانید که برخی از ویژگی های این نرم افزار به افراد کمک می کند تا وزن زیادی کاهش دهند و برخی از ویژگی ها به هیچ وجه به کاهش وزن کمک نمی کنند. بنابراین، هنوز هم می توانید تغییرات را براساس ارزش آنها رتبه بندی کنید.

درک ارزش هر تغییر بیشتر اوقات از تجربه به عنوان یک توسعه دهنده و از انجام تحقیقات صحیح با کاربران برای پیدا کردن آنچه که بیشتر به آن‌ها کمک می‌کند، حاصل می‌شود.

احتمال ارزش و ارزش بالقوه

ارزش در واقع از دو عامل تشکیل شده است: احتمال ارزش (چقدر احتمال دارد که این تغییر به کاربر کمک کند) و مقدار بالقوه (این تغییر چقدر به کاربر در زمان هایی که به آن نیاز دارد کمک می کند)

برای مثال :

- یک ویژگی که می تواند جان کسی را نجات دهد، حتی اگر شانس مورد نیاز بودنش یک در میلیون باشد، هنوز یک ویژگی بسیار ارزشمند است و از ارزش بالقوه بالایی (نجات زندگی) برخوردار است، حتی اگر از احتمال کمی برخوردار باشد. مثال دیگر، در یک برنامه محاسباتی ممکن است ویژگی اضافه کنید که به افراد نابینا کمک می کند شماره ها را در سیستم وارد کنند. فقط درصد کمی از افراد نابینا هستند، اما بدون این ویژگی، آنها به هیچ وجه نمی توانند از نرم افزار شما استفاده کنند. باز هم، این ویژگی با وجود اینکه فقط روی گروه کوچکی از کاربران تأثیر می گذارد (احتمال ارزش کم) ، از ارزش بالقوه بالایی برخوردار است.
- اگر ویژگی وجود دارد که 100٪ کاربران شما را خوشحال میکند، این نیز یک ویژگی ارزشمند است. این یک ارزش بالقوه بسیار جزئی دارد (خوشحال کردن مردم)، اما تعداد بسیار زیادی از کاربران را تحت تاثیر قرار می دهد، بنابراین از احتمال ارزش بالایی برخوردار است.
- از طرف دیگر ، اگر ویژگی خاصی را اجرا کنید که شانس آن در خوشحال کردن مردم یک در میلیون باشد، این خیلی ارزش ندارد. این یک ویژگی با ارزش پتانسیل کم و احتمال ارزش کم است.

بنابراین ، هنگام مد نظر داشتن ارزش ، باید موارد زیر را نیز در نظر بگیرید:

- این تغییر برای چند کاربر(چند درصد)ارزشمند خواهد بود؟
- احتمال اینکه این ویژگی برای کاربر ارزشمند باشد چقدر است؟ یا به بیان دیگر: این ویژگی هر چند وقت یکبار ارزشمند خواهد بود؟
- چه زمانی ارزشمند است، چقدر ارزشمند خواهد بود؟

تعادل آسیب

برخی تغییرات علاوه بر کمکی که می کنند, ممکن است باعث آسیب شوند. به عنوان مثال ، ممکن است برخی از کاربران اگر نرم افزار شما به آنها تبلیغات نشان دهد، اذیت شوند، حتی اگر این تبلیغات به شما به عنوان یک توسعه دهنده کمک کند.

محاسبه ارزش یک تغییر شامل در نظر گرفتن میزان آسیبی است که ممکن است وارد کند و متعادل سازی آن نسبت به کمکی که به همراه دارد.

ارزش داشتن کاربران

ویژگی هایی که هیچ کاربری علاقه به استفاده از آن ندارد هیچ ارزش آنی (لحظه ای) ندارند. اینها می تواند شامل ویژگی هایی باشد که کاربران نمی توانند پیدا کنند، ویژگی هایی که استفاده از آنها بسیار دشوار است یا ویژگی هایی که به راحتی به کسی کمک نمی کنند. آن‌ها ممکن است در آینده ارزشی داشته باشند، اما اکنون هیچ ارزشی ندارند.

این به این معنی است که در اغلب موارد، شما باید واقعا نرم‌افزار خود را راه اندازی کنید تا ارزشمند باشد. تغییری که انجام آن خیلی طولانی می شود در حقیقت می تواند دارای ارزشی معادل صفر باشد چون به موقع راه اندازی نمی‌شود تا به مردم به طور موثر کمک کند. در هنگام تعیین میزان مطلوبیت تغییرات، در نظر گرفتن زمانبندی انتشار می تواند مهم باشد.

تلاش

سنجیدن سعی و تلاش با استفاده از اعداد نسبت به ارزش دادن به آن راحت تر است. معمولاً می توانید تلاش را به عنوان "تعداد معینی از ساعت کار که توسط تعداد معینی از افراد انجام میشود" توصیف کنید. "به اندازه عمر صد نفر" نمونه ای از اندازه گیری عددی است که معمولاً برای تلاش شنیده می شود، نشان دادن ۱۰۰ سال کار توسط یک فرد، ۱ سال کار توسط ۱۰۰ نفر، ۲ سال کار توسط ۵۰ نفر و غیره.

با این حال، حتی با وجود اینکه تلاش می‌تواند به اعداد اندازه گیری شود، اما اندازه گیری آن در موقعیت‌های عملی بسیار دشوار است - شاید هم غیر ممکن. تغییرات می‌توانند هزینه‌های پنهان بسیاری داشته باشند که می توان آن‌ها را به سختی پیش‌بینی کرد، مانند زمانی که برای تعمیر هرگونه اشکالات بوجود آمده ناشی از تغییرات در آینده خرج خواهید کرد. اما اگر شما یک توسعه دهنده نرم افزار باتجربه هستید ، احتمالاً می توانید تغییرات را با توجه به میزان تلاش برای انجام آنها رتبه بندی کنید، حتی اگر اعداد دقیق هر یک را نمی دانید.

هنگام در نظر گرفتن تلاشی که در ایجاد یک تغییر ایجاد می شود ، مهم است که تمام تلاشی را که ممکن است انجام شود، در نظر بگیرید، نه فقط زمانی که می خواهید برنامه نویسی کنید.

چه مدت تحقیق لازم است؟ توسعه دهندگان چقدر باید با یکدیگر در ارتباط باشند ؟ چقدر زمان صرف فکر کردن درباره تغییر می‌کنید؟ به طور خلاصه، هر مقدار زمانی که به یک تغییر مرتبط است، بخشی از هزینه تلاش است.

نگهداری

معادله ای که تاکنون داشته ایم بسیار ساده است ، اما یک عنصر مهم(زمان) را از دست می دهد. شما نه تنها باید یک تغییر را اجرا کنید، بلکه باید آن را با گذشت زمان حفظ کنید. همه تغییرات نیاز به نگهداری دارند. این خیلی واضح است اگر شما برنامه‌ای می نویسید که مالیات مردم را پرداخت میکند، باید هر سال آن را برای قوانین مالیاتی جدید به روز رسانی کنید. اما حتی تغییراتی که به نظر نمی رسد بلافاصله هزینه نگهداری طولانی مدت داشته باشند نیز هزینه ای خواهند داشت, حتی اگر این هزینه فقط نیاز به اطمینان از این است که کد هنگام تست در سال آینده همچنان کار می کند.

ما همچنین باید هم در حال حاضر و هم در آینده ارزش را در نظر بگیریم. وقتی ما تغییراتی را در سیستم خود اعمال کنیم ، این به کاربران فعلی ما کمک می کند ، اما ممکن است به همه کاربران آینده ما نیز کمک کند. حتی ممکن است روی تعداد کل کاربران در آینده تاثیر بگذارد ، بنابراین به طور کلی میزانی که نرم افزار به مردم کمک میکند را تغییر میدهد.

برخی ویژگی‌ها حتی در طول زمان نیز تغییر می‌کنند. برای مثال، داشتن یک نرم افزار مالیاتی که قوانین مالیاتی سال ۲۰۰۹ را درک کند در سال‌های ۲۰۰۹ تا ۲۰۱۰ با ارزش است، اما در سال ۲۰۱۱، دیگر چندان ارزشمند نیست. این ویژگی ای است که در طول زمان کم‌ارزش‌تر می‌شود. برخی از ویژگی‌ها نیز در طول زمان با ارزش تر می‌شوند.

بنابراین، با نگاه به این حقیقت، می‌بینیم که تلاش عملا شامل تلاش برای اجرا و تلاش برای نگهداری و ارزش عملا شامل ارزش هم اکنون و هم ارزش در آینده است. در شکل معادله، این به این شکل است:

$$E = E_i + E_m$$

$$V = V_n + V_f$$

E_i

تلاش برای پیاده سازی است.

E_m

تلاش برای نگهداری است.

V_n

مقدار ارزش کنونی.

V_f

مقدار ارزش آینده.

معادله کامل

وقتی همه اجزا در کنار هم قرار میگیرند ، معادله کامل به این شکل است:

$$D = \frac{V_n + V_f}{E_i + E_m}$$

یا به فارسی :

> مطلوبیت یک تغییر مستقیماً با ارزش حال به علاوه ارزش آینده متناسب است و به صورت معکوس متناسب با تلاش برای اجرا به علاوه تلاش برای نگهداری می باشد.

این قانون اصلی طراحی نرم‌افزار است. با این حال، چیزهای بیشتری در مورد آن وجود دارد.

کاهش معادله

"ارزش آینده" و "تلاش برای نگهداری" هر دو به زمان بستگی دارند، وقتی ما معادله آن را در دنیای واقعی اعمال می‌کنیم باعث می شود چیزهای جالبی اتفاق بیفتد. برای نشان دادن اینها ، بیایید وانمود کنیم که می توانیم از پول برای حل معادله استفاده کنیم, هم برای ارزش و هم برای تلاش . "ارزش" با مقدار پولی که این تغییر برای ما به همراه خواهد داشت اندازه گیری خواهد شد. "تلاش" نیز بر اساس اینکه برای اجرای این تغییر چه مقدار پول باید هزینه شود، اندازه گیری می شود. شما نباید در دنیای واقعی به این شکل از این معادله استفاده کنید، اما به خاطر مثال ما، کارها را ساده می کند.

پس اجازه دهید بگوییم ما در جایی که معادله به این شکل است می خواهیم تغییری ایجاد کنیم :

$$D = \frac{\$10.000 + \$1000/day}{\$1.000 + \$100/day}$$

به عبارت دیگر ، اجرای این تغییر 1000 دلار هزینه دارد (تلاش برای اجرا ، پایین سمت چپ) و بلافاصله 10،000 دلار برای ما به دست می آورد (ارزش فعلی ، بالا سمت چپ). سپس ، هر روز پس از آن ، به ما 1000 دلار (ارزش آینده ، بالا سمت راست) می رسد و نگهداری آن 100 دلار (تلاش برای نگهداری ، پایین سمت راست) هزینه می شود.

پس از 10 روز ، ارزش انباشته آینده بالغ بر 10 هزار دلار و تلاش برای نگهداری بالغ بر 1000 دلار است. این برابر با "ارزش فعلی" اصلی و هزینه اجرا بعد از ۱۰ روز است. پس از 100 روز، ارزش آینده بالغ بر 100000 دلار است و تلاش برای نگهداری به 10 هزار دلار می رسد.

پس از 1000 روز ، ارزش کل آینده به 1،000،000 دلار می رسد و تلاش برای نگهداری بالغ بر 100000 $. در این مرحله ، "ارزش فعلی" اصلی و هزینه اجرا در مقایسه بسیار کوچک به نظر می رسند. هرچه زمان می گذرد ، از اهمیت آنها کاسته میشود و در نهایت به طور کامل از اهمیت میشود. این ، بنابراین ، با گذشت زمان ، معادله ما به چنین چیزی کاهش می یابد:

$$D = \frac{V_f}{E_m}$$

و در واقع، تقریباً تمام تصمیمات در طراحی نرم افزار کاملاً به اندازه گیری ارزش تغییر در آینده در مقابل تلاش برای نگهداری تقلیل می یابد. موقعیت‌هایی وجود دارند که در آن‌ها ارزش فعلی و تلاش اجرایی به اندازه کافی بزرگ هستند که در تصمیم‌گیری مهم باشند، اما بسیار نادر هستند. به طور کلی، سیستم‌های نرم‌افزاری آنقدر طولانی نگهداری می شوند که ارزش فعلی و تلاش برای اجرا در تمامی موارد در مقایسه با ارزش بلندمدت آینده و تلاش برای نگهداری، ناچیز است.

آنچه شما انجام می دهید و نمی خواهید

درس اول برای یادگیری در اینجا این است که ما می خواهیم از شرایطی جلوگیری کنیم که برای یک تغییر خاص، تلاش برای نگهداری، در آخر بیشتر از ارزش آینده باشد. به عنوان مثال ، تصور کنید که شما تغییری را اعمال می کنید که در طی پنج روز, تلاش و ارزش آن به این شکل باشد:

روز	تلاش	ارزش
1	$10	$1.000
2	$100	$100
3	$1.000	$10
4	$10.000	$1
5	$100.000	$0.10
مجموع	$111,110	$1111.10

واضح است که این یک تغییر وحشتناک است که هرگز نباید ایجاد می کردید. اگر کارها با همین سرعت ادامه داشته باشند، شما به هیچ وجه قادر به حفظ سیستم نخواهید بود - این سیستم بی نهایت گران خواهد شد و ارزشی که هر روز کسب میکنید صفر دلار خواهد بود.

هر وضعیتی که در آن تلاش برای نگهداری سریع تر از ارزش آن افزایش می‌یابد، شما را به دردسر می‌اندازد، حتی اگر ابتدا خوب به نظر برسد:

روز	تلاش	ارزش
1	$1000	$1000
2	$2000	$2000
3	$4000	$3000
4	$8000	$4000
مجموع	$15,000	$10,000

راه حل ایده آل - و تنها راه برای تضمین موفقیت - طراحی سیستم های شما به گونه ای است که تلاش برای نگهداری با گذشت زمان کاهش یابد و در نهایت صفر شود (یا تا آنجا که ممکن است به آن نزدیک شوید). تا زمانی که می توانید این کار را انجام دهید ، مهم نیست که ارزش در آینده چقدر بزرگ یا کوچک می شود. لازم نیست نگران آن باشید. به عنوان مثال ، این جدول ها شرایط مطلوبی را نشان می دهند:

روز	تلاش	ارزش
1	$1,000	$0
2	$100	$10
3	$10	$100
4	$0	$1,000
5	$0	$10,000
جمع	$1,110	$11,110

روز	تلاش	ارزش
1	$20	$10

روز	تلاش	ارزش
2	$10	$10
3	$5	$10
4	$1	$10
5	$0	$10
جمع	$36	$50

تغییرات با ارزش آینده بالاتر همچنان مطلوب‌تر هستند. تا زمانی که هزینه نگهداری هر تصمیمی در طول زمان به صفر نزدیک می شود، شما در آینده به درد سر نخواهید افتاد.

از نظر تئوری ، تا زمانی که ارزش آینده همیشه از تلاش نگهداری بیشتر باشد ، تغییر همچنان مطلوب است. بنابراین ، در صورتی که تلاش برای نگهداری و ارزش آینده هر دو افزایش یابد ، می توانید تغییراتی ایجاد کنید، مادامی که ارزش آینده به اندازه کافی بزرگ باشد و از تلاش برای نگهداری بیشتر شود:

روز	تلاش	ارزش
1	$1	$0
2	$2	$2
3	$3	$4
4	$4	$6
5	$5	$8
جمع	$15	$20

چنین تغییری بد نیست ، اما ایجاد تغییری که تلاش برای نگهداری آن کاهش یابد ، مطلوب تر است، حتی اگر تلاش بیشتری برای اجرای آن داشته باشد. اگر تلاش برای نگهداری کاهش یابد، تغییر با گذشت زمان بیشتر و مطلوب تر می شود. این باعث می شود که گزینه بهتری نسبت به سایر احتمالات باشد.

غالباً ، طراحی سیستمی که از تلاش تعمیر و نگهداری کمتری برخوردار باشد، به پیاده سازی و اجرای بسیار بیشتری نیاز دارد - کارهای طراحی و برنامه ریزی بیشتر لازم دارد. با این حال ، به یاد داشته باشید که تلاش برای اجرا تقریباً همیشه یک فاکتور ناچیز در تصمیم گیری در مورد طراحی است، و بیشتر باید نادیده گرفته شود.

به اختصار:

بسیار مهم است که تلاش برای تعمیر و نگهداری را کاهش دهید تا اینکه تلاش برای اجرا را کاهش دهیم.

این یکی از مهمترین مواردی است که باید در مورد طراحی نرم افزار بدانید.

اما چه چیزی باعث تلاش برای نگهداری می شود؟ چگونه سیستم هایی را طراحی کنیم که تلاش نگهداری آنها به مرور کاهش یابد؟ این موضوع قسمت زیادی از بقیه این کتاب است. اما قبل از رسیدن به آن ، باید کمی بیشتر آینده را بررسی کنیم.

کیفیت طراحی

نوشتن نرم‌افزاری که به یک نفر کمک می‌کند بسیار آسان است. اما نوشتن نرم‌افزاری که به میلیون ها نفر در حال حاضر کمک می‌کند و به انجام این کار در آینده ادامه می‌دهد بسیار دشوارتر است.اما بیشتر تلاش برنامه نویسی کجا انجام می شود و چه زمانی بیشتر کاربران از این نرم افزار استفاده می کنند؟ همین الان، یا در دهه های آینده؟

پاسخ این است که برنامه نویسی های بیشتری در آینده روی پروژه انجام می شود - و در آینده میتواند به کاربران بسیار بیشتری نسبت به زمان حال کمک کند . نرم‌افزار شما باید در آینده رقابت کند و در آینده وجود داشته باشد، و تلاش برای نگهداری و تعداد کاربران آن افزایش خواهد داشت.

زمانی که ما این حقیقت را نادیده می‌گیریم که آینده وجود دارد و کارهایی انجام می‌دهیم که در حال حاضر در حال انجام است، نگهداری نرم‌افزار ما در آینده مشکل می‌شود. زمانی که نگهداری نرم‌افزار مشکل است، سخت است که به مردم کمک کنیم (یکی از اهداف ما در طراحی نرم‌افزار). اگر نمی‌توانید ویژگی‌های جدید را اضافه کنید و مشکلات را حل کنید، در نهایت با یک "نرم‌افزار بد" سر و کار دارید که نمیتواند به کاربران شما کمک کند و پر از باگ است.

این امر ما را به قانون زیر هدایت می‌کند:

> سطح کیفیت طراحی شما باید متناسب با مدت زمان آینده باشد که در آن سیستم شما به مردم کمک خواهد کرد.

اگر در حال نوشتن نرم‌افزاری هستید که تنها برای چند ساعت آینده مورد استفاده قرار خواهد گرفت، مجبور نیستید تلاش زیادی برای طراحی آن کنید. اما اگر نرم‌افزار شما برای ۱۰ سال آینده مورد استفاده قرار گیرد (و این خیلی بیشتر از آن چیزی است که انتظار دارید، حتی اگر فکر می‌کنید فقط برای ۶ ماه آینده مورد استفاده قرار خواهد گرفت)، پس باید کار زیادی را برای آن انجام دهید. زمانی که شک دارید، نرم‌افزار خود را طوری طراحی کنید که به مدت طولانی مورد استفاده قرار گیرد: خود را زندانی روش های دیگران نکنید، انعطاف‌پذیر باشید، هیچ تصمیمی نگیرید که دیگر نتوانید تغییر دهید، و به طراحی توجه زیادی کنید.

پیامدهای غیرقابل پیش بینی

بنابراین، هنگام طراحی نرم افزار، آینده باید تمرکز اصلی ما باشد. با این حال، یکی از مهمترین مواردی که باید در مورد هر نوع مهندسی بدانید این است:

> **چیزهایی در مورد آینده وجود دارد که شما نمی‌دانید.**

در حقیقت، وقتی نوبت به طراحی نرم‌افزار می‌رسد، شما نمی‌توانید بیشتر چیزها را در مورد آینده بدانید.

> **رایج‌ترین و فاجعه بارترین خطای برنامه نویسان، پیش‌بینی چیزی در مورد آینده است که در واقع به طور قطعی نمی‌توانند بدانند.**

به عنوان مثال ، تصور کنید که یک برنامه نویس در سال 1985 یک نرم افزار نوشته که دیسک های فلاپی خراب را درست می کند. نمی‌تواند چیز دیگری را تعمیر کند-تک تک قطعات آن کاملاً به نحوه کار فلاپی دیسک ها وابسته بود.

این نرم افزار اکنون منسوخ شده است، زیرا کسی دیگر از دیسک های فلاپی استفاده نمی کند. این برنامه نویس پیش بینی کرد "مردم همیشه از فلاپی دیسک استفاده خواهند کرد"-چیزی که او واقعاً نمی توانست بداند.

ممکن است بشود آینده کوتاه‌مدت را پیشبینی کرد، اما آینده بلند مدت تا حد زیادی ناشناخته است. اما بلند مدت برای ما مهم‌تر از کوتاه‌مدت است، زیرا تصمیمات طراحی ما عواقب بیشتری در دوره طولانی مدت خواهند داشت.

☆اگر سعی در پیش بینی آینده نداشته باشید، در ایمن ترین حالت قرار دارید , به جای آن تمام تصمیمات طراحی خود را براساس اطلاعات زمان حاضر مشخص کنید.

اکنون ممکن است دقیقاً برعکس آنچه در این فصل تاکنون گفته ایم به نظر برسد، اما اینگونه نیست. آینده مهمترین نکته ای است که باید در تصمیم گیری های طراحی مورد توجه قرار گیرد. اما بین طراحی به روشی که امکان تغییرش در آینده را فراهم کند و تلاش برای پیش بینی آینده تفاوت وجود دارد.

به عنوان مثال، بگذارید بگوییم که شما یک گزینه ساده بین غذا خوردن و گرسنگی کشیدن دارید. برای انجام چنین انتخابی لازم نیست آینده را پیش بینی کنید - می دانید که غذا خوردن تصمیم بهتری است. چرا؟ زیرا همین الان شما را زنده نگه می دارد و زنده بودن آینده ای بهتر از مردن را رقم می زند. آینده مهم است و ما می خواهیم آن را در تصمیمات خود در نظر بگیریم. ما اکنون غذا خوردن را انتخاب می کنیم زیرا آینده بهتری ایجاد می کند. اما آینده لازم نیست پیش بینی شود - لازم نیست که جمله خاصی را بیان کنیم مانند "من الان غذا می خورم تا فردا زنده بمانم. "مهم نیست که فردا چه اتفاقی بیفتد، اگر شما الان غذا بخورید بهتر از این است که گرسنه بمیرید.

به همین ترتیب، در طراحی نرم‌افزار ما می‌توانیم تصمیم‌های خاصی را براساس اطلاعاتی که در حال حاضر داریم، به منظور ایجاد آینده‌ای بهتر (کاهش تلاش نگهداری و افزایش ارزش)، بدون نیاز به پیش‌بینی دقیق اتفاقاتی که در آن آینده رخ می‌دهد، بگیریم.

موارد استثنایی محدودی وجود دارد که بعضی اوقات می دانید دقیقاً چه اتفاقی قرار است در آینده کوتاه بیفتد و می توانید بر اساس آن تصمیم بگیرید. اما اگر می خواهید این کار را انجام دهید ، باید در مورد آینده بسیار مطمئن باشید و باید نزدیک باشد. مهم نیست که چقدر باهوش باشید، به هیچ وجه امکان پیش بینی دقیق آینده های بلند مدت وجود ندارد.

بیایید مثالی خارج از حوزه برنامه نویسی بزنیم: CD هایی که در سال 1979 برای جایگزینی نوارهای کاست به عنوان روش اصلی گوش دادن به موسیقی طراحی شده اند.چه کسی می توانست پیش بینی کند که 20 سال بعد ، DVD ها در همان اندازه و شکل ساخته شوند تا تولید کنندگان بتوانند درایوهای CD / DVD را برای رایانه ها بسازند؟

به همین دلیل است که ، در هر نوع مهندسی - از جمله زمینه توسعه نرم افزار - "اصول راهنما" داریم. اینها قوانین خاصی هستند که وقتی از آنها پیروی می کنیم ، بدون توجه به آنچه در آینده اتفاق می افتد ، کارها را به خوبی ادامه می دهند. این ها همان قوانین طراحی نرم افزار هستند- "اصول راهنمای" ما به عنوان طراحان.

پس بله، مهم است که به یاد داشته باشید که آینده وجود خواهد داشت. اما این به این معنا نیست که شما باید آن آینده را پیش‌بینی کنید. در عوض، توضیح می‌دهد که چرا شما باید با توجه به قوانین و مقررات این کتاب تصمیم بگیرید - چون آن‌ها در آینده به یک نرم‌افزار خوب منجر می‌شوند، مهم نیست که آینده چه چیزی را به ارمغان می‌آورد.

حتی پیش بینی همه روشهایی که ممکن است یک قاعده یا قانون خاص در آینده به شما کمک کند امکان پذیر نیست - اما این به شما کمک خواهد کرد ، و خوشحال خواهید شد که از این قانون در کار خود استفاده کرده اید. شما می توانید با قوانین و حقایقی که در اینجا می خوانید مخالفت کنید. لطفاً به نتیجه گیری خود درباره آنها بپردازید. اما باید به شما هشدار داده شود که اگر آنها را دنبال نکنید ، احتمالاً در جایی در پایین مسیر به مشکلی بر خواهید خورد، در آینده ای که نمی توانید پیش بینی کنید.

فصل 6

تغییر

اکنون که اهمیت آینده را فهمیدیم و فهمیدیم مواردی وجود دارد که نمی دانیم و نمی توانیم در مورد آن بدانیم، چه چیزی را می توان در مورد آینده فهمید؟ خوب ، یک چیزی که می توانید از آن اطمینان داشته باشید این است که با گذشت زمان ، محیط نرم افزار شما تغییر می کند. هیچ چیز برای همیشه ثابت نمی ماند. این بدان معنی است که نرم افزار شما باید تغییر کند تا بتواند خود را با محیط اطراف سازگار کند.

این ما را قانون تغییر می رساند:

هرچه برنامه شما برای مدت طولانی‌تری فعال باشد، احتمال تغییر در هر قطعه از آن بیشتر میشود.

در حالی که بسوی آینده بی‌نهایت می‌روید، به سمت یک احتمال ۱۰۰ % می‌روید که هر قطعه از برنامه شما باید تغییر کند. پنج دقیقه بعد احتمالا هیچ بخشی از برنامه شما تغییر نخواهد کرد.در 10 روز آینده ، ممکن است قطعه کوچکی از آن تغییر کند. در 20 سال آینده، احتمالاً اکثریت آن (اگر نه همه آن) باید تغییر کند.

پیش بینی اینکه در آینده دقیقاً چه چیزی تغییر خواهد کرد دشوار است. شاید شما برنامه ای برای اتومبیل های 4 چرخ نوشتید، اما شاید در آینده همه با کامیون های 18 چرخ رانندگی کنند. شاید شما برنامه ای برای دانش آموزان دبیرستان نوشتید، اما تحصیلات دبیرستان چنان بد خواهد شد که دانش آموزان دیگر نمی توانند آن را درک کنند.

نکته این است که لازم نیست سعی کنید که پیش‌بینی کنید چه چیزی تغییر خواهد کرد؛ شما فقط باید بدانید که همه چیز تغییر خواهد کرد. نرم‌افزار خود را بنویسید به طوری که تا حد معقول انعطاف‌پذیر باشد، و شما قادر خواهید بود با هر نوع تغییرات آتی سازگار شوید.

تغییر در یک برنامه دنیای واقعی

بیایید برخی از داده ها را در مورد چگونگی تغییر یک برنامه دنیای واقعی با گذشت زمان بررسی کنیم. صدها فایل در این برنامه خاص وجود دارد، اما جزئیات هر پرونده در این صفحه نمی گنجد ، بنابراین چهار فایل به عنوان نمونه انتخاب شده است. جزئیات مربوط به این پرونده ها در جدول 5-1 آورده شده است.

جدول 5-1. تغییر در فایل ها با گذشت زمان

	فایل 1	فایل 2	فایل 3	فایل 4
دوره آنالیز شده	5سال / 2ماه	8سال/ 3 ماه	13 سال / 3 ماه	13 سال / 4 ماه
خطهای نوشته شده	423	192	227	309
خطهای بدون تغییر	271	101	4	8
خطهای موجود	664	948	388	414
مقدار رشد	241	756	161	105

	فایل 1	فایل 2	فایل 3	فایل 4	
تعدا دفعات تغییر	47	99	194	459	
خط های اضافه شده	396	1,026	913	3,828	
خط های حذف شد	155	270	752	3,723	
خط های اصلاح شده	124	413	1,382	3,556	
کل تغییرات	675	1,709	3,047	11,107	
نسبت تغییر	1.6x	8.9x	13x	36x	

در این جدول:

دوره آنالیز شده

دوره زمانی که فایل وجود داشته است.

خط های اصلی

چه تعداد خط در فایل وجود داشته که در ابتدا نوشته شده است.

خط های بدون تغییر

چند خط از ابتدا تا به حال ثابت مانده است.

خط های موجود

در پایان دوره تجزیه و تحلیل ، اکنون چند خط در فایل وجود دارد.

مقدار رشد

تفاوت بین "خط های اصلی" و "خط های موجود"

تعداد دفعات تغییر

تعداد کل دفعاتی که یک برنامه نویس مجموعه ای از تغییرات را در پرونده ایجاد کرده است (جایی که یک مجموعه از تغییرات شامل تغییر در بسیاری از خطوط است). معمولاً یک مجموعه تغییرات نمایانگر یک رفع اشکال ، یک ویژگی جدید و غیره است.

خط های اضافه شده

چند بار، در طول تاریخ فایل، یک خط جدید اضافه شد.

خط های حذف شد

چند بار، در طول تاریخ فایل، یک خط موجود حذف شده است.

خط های اصلاح شده

چند بار، در طول تاریخ پرونده، یک خط موجود تغییر داده شده است (اما اخیراً اضافه نشده یا حذف نشده است).

کل تغییرات

مجموع "خطوط اضافه شده" ، "خطوط حذف شده" و "خطوط اصلاح شده" برای آن پرونده به حساب می آید.

نسبت تغییر

"تغییرات کلی" چقدر بزرگ‌تر از "خطوط اصلی" هستند.

هنگامی که ما در توضیحات بالا به "خطوط" مراجعه می کنیم، این شامل هر سطر در فایل ها است: کد، نظرات، مستندات و خطوط خالی. اگر بخواهید تجزیه و تحلیل را بدون شمارش نظرات ، مستندات و خطوط خالی انجام دهید ، تفاوت عمده ای که مشاهده خواهید کرد این است که تعداد "خطوط بدون تغییر" به نسبت تعداد دیگر بسیار کوچکتر خواهد شد. (به عبارت دیگر ، خطوط بدون تغییر تقریباً همیشه نظرات ، مستندات یا خطوط خالی هستند).

مهمترین چیزی که از این جدول باید فهمید این است که تغییرات زیادی در یک پروژه نرم افزاری اتفاق می افتد. با گذشت زمان بیشتر و بیشتر احتمال دارد که هر خط خاصی از کد تغییر کند، اما شما نمی توانید دقیقاً پیش بینی کنید چه چیزی تغییر می کند، چه زمانی تغییر می کند یا اینکه چه مقدار باید تغییر کند. هر یک از این چهار فایل به روش‌های بسیار متفاوتی تغییر کرده‌است (شما می‌توانید این را ببینید)، اما همه آن‌ها مقدار قابل‌توجهی تغییر کرده اند.

چیزهای جالب دیگری هم در مورد اعداد وجود دارد، مانند:

- با نگاهی به مقیاس تغییر، می بینیم که برای تغییر هر پرونده بیشتر از نوشتن آن کار شده است. بدیهی است که شمارش خط برآورد کاملی از میزان واقعی کار نیست ، اما درک کلی به ما می دهد. گاهی اوقات نسبت تغییر بسیار زیاد است - به عنوان مثال، فایل 4 ، 36 برابر بیشتر از خطوط اصلی تغییر داشته است.
- تعداد خطوط تغییر یافته در هر پرونده در مقایسه با تعداد "خطوط اصلی" آن کم و در مقایسه با تعداد "خطوط اکنون" آن حتی کمتر است.
- تغییرات زیادی می تواند برای یک فایل اتفاق بیفتد حتی اگر با گذشت زمان فقط کمی بزرگتر شود. به عنوان مثال ، پرونده 3 در طول 13 سال فقط 161 خط رشد کرد ، اما در طول آن تعداد کل تغییرات به 3047 خط رسید.
- تعداد کل شمارش همیشه بزرگ‌تر از تعداد خطوط در حال حاضر است. به عبارت دیگر، احتمال تغییر یک خط در یک فایل بیشتر از نگهداری و داشتن آن خط است، به خصوص زمانی که فایل برای مدت طولانی موجود باشد.
- در پرونده 3 ، تعداد خطوط اصلاح شده بیشتر از تعداد خطوط موجود در فایل اصلی به علاوه تعداد خطوط اضافه شده است. خطوط آن پرونده بیشتر از اضافه شدن خطوط جدید اصلاح شده است. به عبارت دیگر ،

برخی از سطرهای آن پرونده بارها و بارها تغییر کرده اند. این در پروژه هایی با طول عمر طولانی معمول است.

نکات فوق همه مواردی نیست که می توان در اینجا آموخت - تجزیه و تحلیل بسیار جالب تری وجود دارد که می توان روی این اعداد انجام داد. به شما توصیه می شود این داده ها را جستجو کنید (یا اعداد مشابهی را برای پروژه خود بسازید) و ببینید چه چیزهای دیگری می توانید یاد بگیرید.

☆یک تجربه یادگیری خوب دیگر، مرور تاریخچه تغییرات ایجاد شده در یک فایل خاص است. اگر سابقه هر تغییر در فایل ها را در برنامه خود ثبت کرده اید و یک فایل دارید که مدت هاست وجود ندارد ، سعی کنید هر تغییری را که در طول عمر آن انجام شده است ، مشاهده کنید. به این فکر کنید که آیا هنگام نوشتن پرونده می توانستید آن تغییر را پیش بینی کنید یا خیر و در نظر بگیرید که آیا برای ساده تر شدن تغییرات می توان فایل را در ابتدا بهتر نوشت؟ به طور کلی، سعی کنید هر تغییری را درک کنید و ببینید آیا می توانید با این کار چیز جدیدی در مورد توسعه نرم افزار یاد بگیرید.

سه نقص

سه اشتباه گسترده وجود دارد که طراحان نرم افزار هنگام تلاش برای کنار آمدن با قانون تغییر مرتکب می شوند، که به ترتیب میزان شایع بودن آنها در اینجا ذکر شده است:

1. نوشتن کدی که مورد نیاز نیست
2. آسان نساختن کد برای تغییر
3. بیش از حد عام بودن

نوشتن کدی که مورد نیاز نیست

امروزه یک قانون مشهور در طراحی نرم افزار وجود دارد به نام "You Ain't Gonna Need It" یا به اختصار YAGNI. اساساً ، این قانون بیان می کند که شما نباید کد را قبل از نیاز واقعی آن بنویسید. این یک قانون خوب است ، اما نامش اشتباه است. در واقع ممکن است در آینده به کد نیاز داشته باشید ، اما از آنجا که نمی توانید آینده را پیش بینی کنید ، هنوز نمی دانید که کد باید چگونه کار کند. اگر هم اکنون آن را بنویسید، قبل از اینکه به آن احتیاج داشته باشید ، مجبور خواهید شد که برای استفاده واقعی آن را دوباره طراحی کنید. بنابراین در زمان طراحی مجدد خود صرفه جویی کنید و به راحتی صبر کنید تا قبل از نوشتن کد ، به آن نیاز پیدا کنید.

خطر دیگر نوشتن کد قبل از نیاز داشتن به آن کد این است که کد استفاده نشده تمایل به ایجاد "پوسیدگی بیت" دارد. از آنجا که کد هرگز اجرا نمی شود ، ممکن است به آرامی با بقیه سیستم شما همگام شود و در نتیجه اشکال ایجاد کند، و شما هرگز نخواهید فهمید. سپس ، وقتی شروع به استفاده از آن کردید ، باید وقت خود را برای رفع اشکال آن صرف کنید. یا حتی بدتر ، ممکن است به کدی که قبلاً استفاده نشده اعتماد داشته باشید و آن را بررسی نکنید و ممکن است برای کاربران اشکال ایجاد کند. در واقع ، این قانون در اینجا باید گسترش یابد و به شرح زیر باشد:

کد را تا زمانی که واقعاً به آن احتیاج ندارید، ننویسید و کدی را که استفاده نمی شود حذف کنید.

یعنی شما همچنین باید از هر کدی که دیگر نیازی به آن نیست خلاص شوید. در صورت نیاز مجدد ، می توانید همیشه بعداً آن را اضافه کنید.

دلایل زیادی وجود دارد که مردم فکر می کنند باید کدها را قبل از اینکه لازم باشد بنویسند، یا کدی را که استفاده نمی شود نگه دارند. اول از همه ، برخی از مردم معتقدند که می توانند با برنامه نویسی هر ویژگی که ممکن است کاربر به آن نیاز داشته باشد، قانون تغییر را دور بزنند. سپس ، آنها فکر می کنند که در آینده برنامه نباید تغییر کند یا بهبود یابد. اما این اشتباه است. نوشتن سیستمی که هرگز تغییر نخواهد کرد ، تا زمانی که این سیستم به کاربران خود خدمات ارائه دهد ، امکان پذیر نیست.

برخی دیگر بر این باورند که با انجام کار بیشتر در حال حاضر خود را در آینده نجات میدهند. در برخی موارد، این فلسفه کار می‌کند، اما نه وقتی که در حال نوشتن کد هستید که مورد نیاز نیست. حتی اگر این کد در آینده مورد نیاز باشد، شما قطعا باید وقت خود را برای طراحی مجدد آن صرف کنید، پس واقعا دارید زمان را هدر می‌دهید.

نوشتن کد غیر ضروری : یک مثال از دنیای واقعی

روزگاری ، یک توسعه دهنده - بیایید او را Max بنامیم - به اشتباه تصور کرد که می تواند این قانون را نادیده بگیرد. در برنامه وی جعبه های کشویی وجود داشت که کاربران می توانستند مقداری را انتخاب کنند . هر شرکتی که از این برنامه استفاده می کند می تواند لیستی از گزینه های نمایش داده شده در هر کادر کشویی را شخصی سازی کند. برخی از شرکت ها ممکن است بخواهند انتخاب ها نام رنگها باشد. دیگران ممکن است بخواهند آنها نام شهرها باشند. آنها می توانند هر چیزی باشند. بنابراین، لیست گزینه های معتبر باید جایی ذخیره شود که هر شرکت بتواند آن را اصلاح کند.

کار واضحی که باید انجام شود فقط ذخیره کردن لیست مقادیر بود. به هر حال ، این تنها چیزی است که لازم بود. اما ماکس تصمیم گرفت دو چیز را ذخیره کند: لیست مقادیر ، و همچنین اطلاعاتی درباره اینکه آیا هر مقدار در حال حاضر "فعال" است - یعنی اگر کاربران در حال حاضر می توانند آن مقدار را انتخاب کنند یا به طور موقت غیرفعال شده است.

با این حال، ماکس هیچ برنامه‌ای برای استفاده از اطلاعات در مورد این که آیا هر حوزه فعال است یا خیر، ننوشته است. تمام انتخاب‌ها فعال بودند، تمام مدت، صرف‌نظر از آنچه که داده‌های ذخیره‌شده بیان می‌کردند. او مطمئن بود که در شرف نوشتن یک کد برای استفاده از اطلاعات "فعال" است.

چندین سال گذشت و کدی برای مدیریت داده های "فعال" نوشته نشد. در عوض ، داده ها فقط در آنجا ثبت شده بودند ، و برای کاربران گیج کننده بودند و اشکال ایجاد می کردند. بسیاری از مشتریان و توسعه دهندگان برای مکس نوشتند که چرا هیچ اتفاقی نیفتاده است زیرا آنها لیستی از مقادیر را به صورت دستی ویرایش می کنند و گزینه های غیرفعال را تنظیم می کنند. یکی از توسعه دهندگان به نادرست تصور کرد که قسمت "فعال" در حال استفاده است و کدی را نوشت که از آن استفاده می کند ، حتی اگر بقیه سیستم از آن استفاده نکرده باشند. این موضوع به مشتری منتقل شد و آنها شروع به گزارش اشکالات عجیبی کردند که برای ردیابی آنها کار زیادی لازم بود.

سرانجام، برخی از توسعه دهندگان آمدند و گفتند ، "امروز ما توانایی غیرفعال کردن گزینه ها را اجرا خواهیم کرد!" با این حال ، آنها کشف کردند که فیلد "فعال" کاملاً متناسب با نیازهای آنها طراحی نشده است ، بنابراین آنها مجبور شدند برای اجرای ویژگی خودشان کمی طراحی مجدد انجام دهند.

نتیجه: چندین اشکال ، گیجی زیاد و کارهای اضافی برای توسعه دهنده ای که در نهایت واقعاً به کد احتیاج داشت. و این یک تخلف نسبتاً جزئی از قانون بود! تخلفات شدید می تواند عواقب قابل ملاحظه ای بدتر از جمله مهلت های از دست رفته ، فاجعه های بزرگ و حتی احتمالاً تخریب پروژه نرم افزاری شما به همراه داشته باشد.

نتیجه خالص: چندین اشکال ، گیجی زیاد و کارهای اضافی,همه برای توسعه دهنده ای که در نهایت واقعاً به کد احتیاج داشت. و این یک تخلف نسبتاً جزئی از قانون بود! تخلفات شدید می تواند عواقب بدتر قابل ملاحظه ای از جمله مهلت های از دست رفته ، فاجعه های بزرگ و حتی احتمالاً تخریب پروژه نرم افزاری شما به همراه داشته باشد.

آسان نکردن کد برای تغییر

یکی از قاتلین بزرگ پروژه‌های نرم‌افزاری، چیزی است که ما آن را "طراحی سخت" می‌نامیم. دو راه برای به دست آوردن یک طرح سخت وجود دارد :

1. فرضیات بسیار زیادی در مورد آینده داشته باشید.

2. کد را بدون طراحی کافی بنویسید.

مثال : ایجاد فرضیات بسیار در مورد آینده

یک اداره دولتی - اجازه دهید آن را بیمارستان بنامیم - می‌خواهد یک برنامه بسازد . ما این برنامه را "سیستم بهداشتی" می نامیم. قبل از ساخت این سیستم ، دولت تصمیم می گیرد سندی بنویسد که دقیقاً نحوه اجرای کل سیستم را بیان کند. یک سال را صرف نوشتن این سند می‌کند و هر تصمیم واحدی را در مورد کل سیستم در طول این زمان می‌گیرد.

سپس توسعه دهندگان سه سال را صرف نوشتن سیستم طبق این سند می کنند. وقتی کار می کنن ، می فهمند که طرح موجود در سند متناقض و ناقص است و سخت اجرا می شود. اما یک سال تمام طول کشید تا نوشته شود - توسعه دهندگان نمی‌توانند یک سال دیگر صبر کنند تا آن را بازبینی کنند. پس آن‌ها سیستم را اجرا می‌کنند و تا آنجا که می‌توانند از سند پیروی می‌کنند. سیستم تکمیل شده و برای اولین بار در اختیار کاربران قرار می گیرد. با این حال، وضعیت بیمارستان در چهار سال گذشته به طرز چشمگیری تغییر کرده است و هنگامی که کاربران شروع به استفاده از سیستم بهداشت و درمان می کنند، متوجه میشوند که چیزی کاملا متفاوت می خواهند.اما این سیستم از صدها هزار خط کد درست شده‌است که همه با توجه به سند طراحی شده‌اند - بدون ماهها یا سالها تلاش به راحتی قابل تغییر نیست.

بنابراین بیمارستان شروع به نوشتن سند جدیدی برای یک سیستم جدید می کند و روند کار از ابتدا شروع می شود.

اشتباه بیمارستان تلاش برای پیش بینی آینده بود. آنها فرض کردند که هر تصمیمی که در این سند بگیرند برای کاربران واقعی معتبر است و با تکمیل سیستم همچنان معتبر خواهند بود. وقتی آینده واقعی فرا رسید ، اصلاً شبیه آن چیزی نبود که آنها پیش بینی کرده بودند و سیستم آنها یک شکست چند میلیون دلاری بود.

راه حل بهتر این بود که فقط یک ویژگی یا یک مجموعه کوچک از ویژگی ها مشخص شود و بلافاصله از توسعه دهندگان بخواهید آن را پیاده سازی کنند. پس از آن می توانستند هنگام فرآیند توسعه با کاربر ارتباط برقرار کنند و آزمایشات کاربری را اجرا کنند. هنگامی که اولین مجموعه از ویژگی ها اضافه و منتشر شد، آنها می توانستند یکی یکی روی ویژگی های اضافی کار کنند تا اینکه سرانجام سیستمی داشتند که به خوبی طراحی شده بود و کاملاً نیازهای کاربران را تأمین می کرد.

مثال : کد بدون طراحی کافی

از یک توسعه دهنده خواسته می شود تا برنامه ای را ایجاد کند که افراد بتوانند از آن برای پیگیری وظایفی که باید انجام دهند استفاده کنند. برای ایجاد یک "وظیفه" جدید در سیستم ، کاربران یک فرم را با برخی اطلاعات ، مانند خلاصه ای کوتاه از کار و مدت زمان انجام آن ، پر می کنند. این داده ها را در یک پایگاه داده ذخیره می کند. سپس ، آنها می توانند با گذشت زمان در مورد پیشرفت خود در این کار یادداشت برداری کنند و در نهایت یادآوری کنند که کار را به پایان رسانده اند.

فیلدی به نام "وضعیت" وجود دارد که نشان می دهد کاربر در انجام کار چقدر فاصله دارد. مقادیر این قسمت "کار انجام نشده" ، "در حال انجام" ، "در انتظار" و "کامل" است. وقتی قسمت وضعیت دارای مقدار "کار انجام نشده" باشد ، می تواند فقط به "در حال انجام" تغییر کند. وقتی قسمت وضعیت "در حال انجام" است ، می تواند به "در انتظار" یا "کامل" تغییر کند. و وقتی "کامل" باشد ، فقط می تواند به "در حال پیشرفت" تغییر کند.

10 قسمت دیگر در این برنامه با قوانین مشابه وجود دارد. هر کدام از آنها اطلاعات متفاوتی در مورد کار دارند (به عنوان مثال ، اینکه چه کسی تعیین شده است برای این وظیفه، آخرین مهلت آن و غیره).

برای اجرای این قوانین ، توسعه دهنده یک کد بسیار طولانی و مداوم و بدون ساختار را در یک پرونده می نویسد. او هر فیلد را با کد سفارشی که مخصوص آن قسمت است تأیید می کند. به عنوان مثال ، هر بار که نیاز به بررسی وضعیت "کامل" دارد ، وی به معنای واقعی کلمه "کامل" را در کد می نویسد. همچنین، کد برای استفاده مجدد نوشته نشده است. در مواردی که برنامه دارای زمینه های مشابه است ، توسعه دهنده کد را برش می زند و سپس آن را برای قسمت جدید کمی تغییر می دهد.

کد کار می‌کند. این پرونده ۳۰۰۰ خط دارد. تقریبا فاقد یک طرح است.

چندین ماه بعد ، این توسعه دهنده پروژه را ترک می کند.

یک توسعه دهنده جدید آمده و برای حفظ این پروژه به او محول شده است. و به سرعت متوجه می‌شود که تغییر این کدها سخت است - اگر او یک بخش از آن را تغییر دهد، او همچنین باید بسیاری از بخش‌های دیگر را به همان روش تغییر دهد تا کار را ادامه دهد. برای بدتر کردن اوضاع ، قسمت های مختلف بدون توضیح و سیستم منطقی در اطراف پراکنده شده اند - شما باید هر بار که می خواهید تغییری ایجاد کنید ، به راحتی کل فایل را بخوانید.

مشتریان درخواست ویژگی های جدید را شروع می کنند. در ابتدا، توسعه دهنده جدید تمام تلاش خود را برای پیاده سازی این ویژگی های جدید انجام می دهد. وی حتی کد بیشتری به این پرونده اضافه می کند. در نهایت 5000 خط طول دارد.

در نهایت ، مشتریان شروع به درخواست ویژگی هایی می کنند که با این طرح به سادگی قابل اجرا نیست. آنها می خواهند اطلاعات مربوط به وظایف را از طریق ایمیل ارسال کنند ، اما این کد فقط با یک فرم انجام می شود. همه اینها به طور خاص در مورد نحوه کار فرم طراحی شده اند - هرگز با ایمیل کار نمی کنند.

آنها تمام تلاش خود را می کنند تا در هنگام طراحی جدید ، سایر درخواستهای ویژگی را دنبال کنند ، اما بیشتر وقت خود را صرف طراحی مجدد می کنند.

در بخشی از یک برنامه خاص ، کاربر یک فرم را پر کرد و برنامه چند صد ایمیل ارسال کرد. این قسمت از برنامه بسیار کند بود. کاربر فرم را ارسال می کند و برنامه برای مدت زمان طولانی در آنجا می نشیند و تمام پیام ها را می فرستد.

قاعده مورد استفاده برای جلوگیری از طراحی سخت این است:

کد باید بر اساس آنچه اکنون می دانید طراحی شود، نه بر اساس آنچه فکر می کنید در آینده اتفاق خواهد افتاد.

فقط بر اساس نیازهای فوری و شناخته شده تان طراحی کنید، بدون اینکه احتمال نیازهای آینده را کنار بگذارید. اگر به طور حتم می دانید که برای انجام X و فقط X به سیستم نیاز دارید، همین حالا آن را برای انجام X طراحی کنید. این ممکن است کارهای دیگری را در آینده انجام دهد که X نیستند، و شما باید این را در ذهن داشته باشید، اما در حال حاضر سیستم باید فقط X را انجام دهد.

طراحی به این شکل ، به کوچک نگه داشتن تغییرات فردی نیز کمک می کند. وقتی فقط باید یک تغییر کوچک انجام دهید، آسان است که یک طراحی واقعی روی آن انجام دهید.

این بدان معنا نیست که برنامه ریزی بد است. مقدار مشخصی از برنامه ریزی در طراحی نرم افزار بسیار ارزشمند است. اما حتی اگر برنامه های مفصلی را هم ننویسید، تا زمانی که تغییرات شما همیشه کوچک بوده و کد شما به راحتی با آینده ناشناخته سازگار باشد، شما مشکلی نخواهید داشت.

عمومی بودن بیش از حد

هنگامی که با این واقعیت روبرو می شویم که کد آنها در آینده تغییر خواهد کرد، برخی از توسعه دهندگان سعی می کنند با طراحی یک راه حل کاملاً عمومی (به اعتقاد آنها) این مشکل را با هر وضعیت ممکن در آینده تطبیق دهند. ما این را "مهندسی بیش از حد" می نامیم.

فرهنگ لغت مهندسی بیش از حد را ترکیبی از "over" (به معنی "بیش از حد") و "engineer" (به معنی "طراحی و ساخت") تعریف می کند. بنابراین ، از نظر فرهنگ لغت ، این به معنای طراحی یا ساخت بیش از حد مناسب شرایط شما است.

صبر کنید - طراحی یا ساخت بیش از حد ؟ چه چیزی "بیش از حد" است؟ مگر طراحی چیز خوبی نیست؟

خوب ، بله ، بیشتر پروژه ها باید از طراحی بیشتری استفاده کنند، همانطور که در "مثال: کد بدون طراحی کافی" در صفحه 37 دیدیم. اما هر چند وقت یکبار، شخصی به آن می شود زیاده روی میکند- به نوعی مانند ساختن لیزر برای از بین بردن مورچگان است. لیزر یک ابزار مهندسی شگفت انگیز است. اما هزینه آن بسیار زیاد است، ساخت آن بسیار زمانبر و نگهداری آن بسیار سخت است. آیا می توانید تصور کنید که هنگام خراب شدن باید به آنجا بروید و آن را تعمیر کنید؟

مشکلات دیگری هم با (مهندسی بیش از حد)overengineering وجود دارد:

1. شما نمی توانید آینده را پیش بینی کنید ، بنابراین مهم نیست که راه حل شما چقدر عمومی باشد ، آنقدر عمومی نخواهد بود که بتواند نیازهای واقعی آینده را برآورده کند.

2. هنگامی که کد شما بسیار عمومی است ، از نظر کاربر اغلب به خوبی از پس موارد خاصی بر نمی آید. به عنوان مثال، می توان گفت که شما یک کد را طراحی می‌کنید که با تمام ورودی‌های یک‌سان رفتار می‌کند - همه آن‌ها فقط بایت هستند. بعضی اوقات این کد متن را پردازش می کند و گاهی اوقات تصاویر را پردازش می کند ، اما تنها چیزی که می داند این است که بایت دریافت می کند. به نوعی ، این یک طراحی خوب است: کد ساده ، مستقل ، کوچک و غیره است.

اما سپس مطمئن شوید که هیچ بخشی از کد شما بین تصاویر و متن تفاوت قائل نیست. این خیلی عمومی است. هنگامی که کاربر در یک تصویر بد عبور می کند ، خطایی که دریافت می کند این است: "شما در بایت های بد عبور کردید".باید می گفت: "شما از یک عکس بد عبور کردید" ، اما کد شما آنقدر عمومی است که نمی تواند این موضوع را به کاربر بگوید.

(روش‌های زیادی وجود دارند که یک کده ژنتیکی می‌توانند زمانی کوتاه بیایند که کاربردهای خاص داشته باشند؛ این فقط یک مثال است).

3- خیلی عمومی بودن شامل نوشتن کدهای زیادی است که نیازی نیست ، که ما را به اولین نقص خود بازمی گرداند.

به طور کلی ، وقتی طراحی شما به جای ساده کردن کارها، کارها را پیچیده تر می کند، شما بیش از حد مهندسی می کنید. این لیزر مداری زندگی فردی را که فقط به نابودی برخی مورچه ها احتیاج داشت، بسیار پیچیده می کند، در حالی که برخی از سموم مورچه ساده با از بین بردن مشکل مورچه، زندگی آن شخص را بسیار ساده می کنند (با فرض اینکه موثر باشد).

سخاوت داشتن در موارد درست، با روش های صحیح، می تواند پایه و اساس یک طراحی نرم افزار موفق باشد. با این حال، بیش از حد عمومی بودن می‌تواند علت پیچیدگی ناگفته، سردرگمی و تلاش نگهداری باشد. قانون اجتناب از این عیب شبیه به قانون اجتناب از طرح‌های سخت است:

فقط به همان اندازه عمومی باشید که می دانید در حال حاضر لازم است.

مثال : عمومی بودن بیش از حد

در یک برنامه مشخص، کاربر یک فرم را پر می کرد و برنامه صدها ایمیل ارسال کرد. کاربر فرم را ارسال می‌کند و برنامه به مدت بسیار طولانی صبر می کند و سپس همه پیام‌ها را ارسال می‌کند. این قسمت از برنامه خیلی کند است.

برای سریعتر کردن این قسمت، توسعه دهندگان تصمیم گرفتند که بلافاصله همه ایمیل ها را ارسال نکنند. در عوض ، پس از ارسال فرم توسط کاربر، با استفاده از قطعه کد از قبل موجود به نام "ارسال کننده ایمیل" ، آنها در پس زمینه ارسال می شوند.

توسعه دهنده ای که شروع به کار بر روی این تغییر کرد، متوجه شد که برخی از شرکت‌ها ممکن است بخواهند از چیزی غیر از"ارسال کننده ایمیل" استفاده کنند. او صدها خط کد نوشت که به مشتریان امکان می دهد سیستم های دیگر را برای انجام کارهای پس زمینه "وصل" کنند .هیچ مشتری تا به حال این درخواست را نکرده بود. توسعه دهنده پیش‌بینی کرده بود که ممکن است کسی این نوع انعطاف‌پذیری را در آینده بخواهد.

سرانجام معمار اصلی این برنامه بر روی این تغییر کار کرد. او همه کدها را برای "اتصال به سیستم‌های دیگر" را حذف کرد، چون هیچ مدرکی وجود نداشت که کاربران آن را بخواهند. بنابراین، هیچ مدرکی وجود نداشت مبنی بر این که این کد در حال حاضر باید عمومی باشد. با برداشتن آن قطعات کد، این تغییر بسیار ساده تر شد.

چهار سال از زمان تغییر اصلی می گذرد و حتی یک مشتری به توانایی اتصال سایر سیستم ها نیاز نداشته است. در واقع ، هیچ دلیلی برای وجود چنین عمومی وجود نداشت.

توسعه و طراحی افزایشی

روشی برای توسعه نرم افزار وجود دارد که به دلیل ماهیت خود از سه نقص جلوگیری می کند ، به نام "توسعه و طراحی افزایشی". این شامل طراحی و ساخت یک سیستم قطعه قطعه و به ترتیب.

آسان تر است که با مثال توضیح دهید. در اینجا چگونگی استفاده از آن برای توسعه یک برنامه ماشین حساب است که نیاز به جمع ، تفریق ، ضرب و تقسیم دارد:

1. سیستمی را برنامه ریزی کنید که فقط جمع و کار دیگری انجام دهد.
2. آن سیستم را پیاده سازی کنید.
3. طراحی سیستم موجود را برطرف کنید تا افزودن ویژگی تفریق آسان باشد.
4. ویژگی تفریق را در سیستم پیاده سازی کنید. اکنون ما سیستمی داریم که فقط جمع و تفریق را انجام می دهد و نه چیز دیگری.
5. دوباره طراحی سیستم را برطرف کنید تا افزودن ویژگی ضرب آسان باشد.
6. ویژگی ضرب را در سیستم پیاده سازی کنید. اکنون ما سیستمی داریم که جمع ، تفریق ، ضرب و غیره را انجام می دهد.

7. دوباره طراحی سیستم را برطرف کنید تا افزودن ویژگی تقسیم آسان باشد. (در این مرحله ، این کار باید کم یا بدون هیچ کوششی باشد ، زیرا ما قبل از اجرای تفریق و ضرب ، طراحی را بهبود بخشیده ایم)

8. ویژگی تقسیم را در سیستم پیاده سازی کنید. اکنون ما سیستمی را داریم که قصد ساخت آن را داریم ، با طراحی عالی که مناسب آن است.

این روش توسعه نسبت به برنامه ریزی کل سیستم از قبل و ساخت یک باره، به زمان و تفکر کمتری نیاز دارد. ممکن است در ابتدا اگر به روشهای دیگر توسعه عادت کرده باشید آسان نباشد، اما با تمرین آسان خواهد شد. قسمت جالب استفاده از این روش تصمیم گیری در مورد نحوه اجرای آن است . به طور کلی ، در هر مرحله ، وقتی به آنجا رسیدید ، باید ساده ترین کارها را انتخاب کنید. ما اول جمع را به این دلیل انتخاب کردیم که در کل از بین چهار عمل ساده ترین کار بود، و دوم تفریق برای اینکه بصورت کاملا منطقی بر اساس جمع ساخته شده. ما احتمالاً می توانستیم ضرب را در جایگاه دوم انتخاب کنیم، زیرا ضرب فقط عمل انجام چندین بار جمع است. تنها چیزی که ما نمی توانستیم دوم انتخاب کنیم تقسیم است، زیرا رفتن از جمع به مرحله تقسیم جهش منطقی بزرگی است که بسیار پیچیده است. از طرف دیگر ، رفتن از ضرب به تقسیم در انتها واقعاً بسیار ساده بود ، بنابراین انتخاب خوبی بود.

حتی ممکن است گاهی اوقات لازم باشد که یک ویژگی واحد را بردارید و آن را به بسیاری از مراحل کوچ ، ساده و منطقی تقسیم کنید تا به راحتی قابل اجرا باشد.

این در واقع ترکیبی از دو روش است: یکی به نام "توسعه افزایشی" و دیگری به نام "طراحی افزایشی". توسعه افزایشی روشی است برای ایجاد یک سیستم کامل با کار در قطعات کوچک. در لیست ما ، هر مرحله که با "پیاده سازی" شروع می شد بخشی از روند توسعه افزایشی بود. طراحی افزایشی به همین ترتیب روشی برای ایجاد و بهبود طراحی سیستم در مراحل کوچک است. هر مرحله که با "اصلاح طرح سیستم" یا "طرح" شروع می شد بخشی از روند طراحی افزایشی بود. توسعه و طراحی افزایشی تنها روش معتبر توسعه نرم افزار نیست ، اما روشی است که به طور قطع از سه نقص ذکر شده در بخش قبلی جلوگیری می کند.

فصل 7

نقص ها و طراحی

متاسفانه، هیچ برنامه نویسی کامل نیست. برنامه نویسان خوب به ازای هر 100 خط کدی که می نویسند تقریباً یک نقص را در برنامه وارد می کنند. بهترین برنامه نویسان، در بهترین شرایط ممکن، به ازای هر 1000 خط کدی که می نویسند، یک نقص وارد می کنند.

به عبارت دیگر، مهم نیست که شما به عنوان یک برنامه نویس چقدر خوب یا بد باشید، مطمئناً هرچه بیشترکد بنویسید، نقایص بیشتری نیز ایجاد خواهید کرد. این به ما اجازه می دهد قانونی را به نام قانون احتمال نقص بیان کنیم:

شانس معرفی عیب و نقص در برنامه شما متناسب با اندازه تغییراتی است که شما به آن ایجاد می کنید.

این مهم است زیرا عیب و نقص ها هدف ما را در کمک به مردم نقض می کند، بنابراین باید از آنها اجتناب شود. همچنین، رفع نقص نوعی نگهداری است. بنابراین، افزایش تعداد نقص ها تلاش ما را برای نگهداری را افزایش می دهد.

با این قانون، بدون نیاز به پیش‌بینی آینده، ما بلافاصله می‌توانیم ببینیم که ایجاد تغییرات کوچک به احتمال زیاد منجر به تلاش برای نگهداری کم‌تر نسبت به تغییرات بزرگ می‌شود.

تغییرات کوچک = نقص های کمتر = نگهداری کمتر

این قانون همچنین گاهی اوقات غیررسمی تر بیان می شود: "اگر کد را اضافه نکنید یا آن را تغییر ندهید ، نمی توانید اشکالات جدیدی را بوجود آورید."

نکته جالب در مورد این قانون این است که به نظر می‌رسد در تضاد با قانون تغییر باشد - نرم‌افزار شما باید تغییر کند، اما تغییر آن نقص‌هایی دارد. این یک تعارض واقعی است و این قوانین را متعادل می‌کند که به هوش شما به عنوان یک طراح نرم‌افزار نیاز دارد. در واقع آن اختلافی است که توضیح می‌دهد چرا ما به طراحی نیاز داریم، و در حقیقت به ما می‌گوید که طرح ایده‌آل چیست:

بهترین طراحی موردی است که با کم‌ترین تغییر در نرم‌افزار امکان تغییر در محیط را فراهم می‌کند.

و این ، خیلی ساده ، خلاصه بسیاری از مواردی است که امروزه در مورد طراحی نرم افزار خوب شناخته شده است.

اگر خراب نباشد ...

بسیار خوب ، بنابراین اگر کد را اضافه یا اصلاح نکنید نمی توانید اشکالاتی را در برنامه خود وارد کنید ، و این یک قانون اصلی در طراحی نرم افزار است. با این حال، یک قاعده مرتبط بسیار مهمی نیز وجود دارد که بسیاری از مهندسان نرم افزار به این شکل شنیده اند، اما گاهی اوقات فراموش می کنند:

هرگز چیزی را "تعمیر " نکنید ، مگر اینکه مشکلی باشد و شواهدی دارید که نشان می دهد مشکل واقعاً وجود دارد.

داشتن شواهدی از مشکلات قبل از رفع آنها مهم است. در غیر این صورت، شما ممکن است در حال توسعه ویژگی‌هایی باشید که مشکل هیچ کسی را حل نمیکنند، و یا ممکن است در حال تعمیر چیزهایی باشید که خراب نیستند.

اگر بدون مدرک مشکلی را برطرف کنید، احتمالاً همه چیز را خراب می کنید. شما در حال وارد کردن تغییراتی در سیستم خود هستید، که نقص های جدیدی را به همراه دارد. و نه فقط این، بلکه شما بیهوده وقت خود را هدر می دهید و به برنامه خود پیچیدگی اضافه می کنید.

بنابراین چه چیزی "مدرک" محسوب می شود؟ فرض کنید پنج کاربر گزارش می دهند که وقتی دکمه قرمز را فشار می دهند، برنامه شما خراب می شود. خوب ، این مدرک کافی است! حتی ممکن است خودتان دکمه قرمز را فشار دهید و متوجه خراب شدن برنامه شوید.

با این حال، فقط به این دلیل که کاربر چیزی را گزارش می کند، به معنای مشکل نیست. گاهی اوقات کاربر به راحتی متوجه نخواهد شد که برنامه شما از قبل دارای برخی ویژگی ها بوده است، بنابراین از شما خواسته است که مورد دیگری را بی جهت پیاده سازی کنید. به عنوان مثال، شما برنامه ای می نویسید که لیستی از کلمات را به ترتیب حروف الفبا مرتب می کند، و یک کاربر از شما میخواهد که یک ویژگی اضافه کنید که لیستی از حروف را به ترتیب حروف الفبا مرتب می کند. برنامه شما قبلاً این کار را انجام داده است. در واقع ، بیشتر از این هم انجام میدهد. اغلب مسئله ب این شکل است

> ☆اگر درخواست های زیادی مانند موارد بالا دریافت کنید، به این معنی است که کاربران نمی توانند به راحتی ویژگی های مورد نیاز را در برنامه شما پیدا کنند. این چیزی است که شما باید اصلاح کنید.

گاهی اوقات یک کاربر گزارش می کند که یک اشکال وجود دارد، در حالی که این برنامه دقیقاً همانطور که شما قصدش را داشتید رفتار می کند. در این مورد، موضوع اکثریت است. اگر تعداد قابل توجهی از کاربران فکر کنند که این رفتار یک اشکال است، این یک اشکال است. اگر فقط یک اقلیت کوچک (مانند یک یا دو) فکر می کند این یک اشکال است، این یک اشکال نیست.

معروف ترین خطا در این زمینه همان چیزی است که ما آن را "بهینه سازی زودرس" می نامیم. به نظر می‌رسد که برخی از توسعه دهندگان دوست دارند که کارها را سریع انجام دهند، اما قبل از اینکه بدانند که کد آنها کند است، زمان را صرف بهینه‌سازی کد خود می‌کنند. این مانند خیریه است که برای افراد ثروتمند غذا می فرستد و می گوید: "ما فقط می خواستیم به مردم کمک کنیم!" غیر منطقی است ، نه؟ آنها در حال حل مشکلی هستند که وجود ندارد.

تنها قسمت‌هایی از برنامه شما که در آن باید نگران سرعت باشید، بخش‌های دقیقی هستند که می‌توانید نشان دهید که باعث ایجاد یک مشکل عملکرد واقعی برای کاربران شما می‌شوند. برای بقیه کد، نگرانی های اصلی انعطاف پذیری و سادگی است ، نه اینکه آن را سریع ادامه دهید.

روش های بی حد و حصر برای نقض این قانون وجود دارد ، اما روش پیروی از آن ساده است: قبل از پرداختن به آن ، شواهد واقعی مبنی بر معتبر بودن یک مشکل بدست آوردید.

خودتان تکرار نکنید

این احتمالاً شناخته شده ترین قانون در طراحی نرم افزار است. این کتاب اولین جایی نیست که این قانون تا به حال ظاهر شده است. اما معتبر است، و بنابراین در اینجا گنجانده شده است:

در هر سیستم خاص ، هرگونه اطلاعات، در حالت ایده آل ، فقط یک بار باید وجود داشته باشد.

بیایید بگوییم شما یک قسمت به نام "رمز عبور" دارید که در 100 صفحه رابط کاربری برنامه شما ظاهر می شود. اگر بخواهید نام فیلد را به "کد عبور" تغییر دهید چه می کنید؟ خوب ، اگر نام فیلد را در یک مکان مرکزی در کد خود ذخیره کرده اید، برای رفع آن نیاز به تغییر یک خط کد است. اما اگر کلمه "رمز عبور" را به صورت دستی در تمام 100 صفحه رابط کاربری نوشتید، برای اصلاح آن باید 100 تغییر ایجاد کنید.

این مورد همچنین در مورد بلوک های کد اعمال می شود. شما نباید در حال کپی کردن و چسباندن بلوک های کد باشید. در عوض ، شما باید از بخشهای مختلف فناوری برنامه نویسی استفاده کنید که به یک قطعه کد اجازه می دهد قطعه دیگری از کد موجود را "استفاده" "فراخوانی" یا "اضافه" کند.

یکی از دلایل خوب پیروی از این قانون ، قانون احتمال نقص است. اگر بتوانیم از کد قدیمی استفاده مجدد کنیم، مجبور نیستیم هنگام اضافه کردن ویژگی های جدید، به همان اندازه کد بنویسیم یا تغییر دهیم ، بنابراین نقص های کمتری را معرفی می کنیم.

همچنین به ما در انعطاف پذیری در طراحی هایمان کمک می کند. اگر لازم باشد نحوه کار برنامه خود را تغییر دهیم، می توانیم به جای اینکه کل برنامه را مرور کرده و تغییرات متعددی را اعمال کنیم ، فقط در یک مکان برخی از کدها را تغییر دهیم.

بسیاری از طراحی های خوب بر اساس این قاعده است. یعنی هرچه بیشتر باعث شوید که کد شما از کد های دیگر "استفاده" کند و اطلاعات را متمرکز کنید, طراحی بهتری بدست می آورید. این قسمت دیگری است که هوش شما در برنامه نویسی نقش دارد.

فصل 8

سادگی

بسیار خوب ، بنابراین اگر هرگز نرم افزار خود را تغییر ندهیم ، می توانیم به طور کامل از نقص جلوگیری کنیم. اما تغییر اجتناب ناپذیر است ، به ویژه اگر بخواهیم ویژگی های جدیدی اضافه کنیم. بنابراین "چیزی را تغییر ندهید" نمی تواند تکنیک نهایی کاهش نقص باشد.

همانطور که در فصل 6 توضیح داده شده است ، اگر می خواهید از نقص در کد خود جلوگیری کنید، کوچک نگه داشتن تغییرات به شما کمک می کند. اما اگر می خواهید فراتر بروید و نقص ها را حتی از تغییرات کوچک خود برطرف کنید، قانون دیگری وجود دارد که می تواند به شما کمک کند. و این فقط نقص را کاهش نمی دهد - کد شما را قابل نگهداری می کند ، افزودن ویژگی های جدید را آسان می کند و درک کلی کد شما را بهبود می بخشد. این قانون سادگی است:

سهولت در نگهداری هر نرم افزار متناسب با سادگی قطعات کد جداگانه آن است.

یعنی هرچه قطعات ساده تر باشند ، به راحتی می توانید در آینده چیزها را تغییر دهید. سهولت کامل در نگهداری غیرممکن است ، اما این هدفی است که شما برای آن تلاش می کنید - تغییر کامل یا کد جدید بی نهایت بدون مشکل. ممکن است متوجه شده باشید که این قانون در مورد سادگی کل سیستم صحبت نمی کند ، فقط قطعات جداگانه. چرا؟

خوب ، یک برنامه رایانه ای به طور متوسط آنقدر پیچیده است که هیچ انسانی نمی تواند یکباره آن را درک کند. فقط می‌توان قسمت هایی آن را درک کرد. بنابراین ما در واقع همیشه یک ساختار پیچیده و بزرگ برای کل برنامه خود داریم. آنچه در این صورت مهم می شود این است که وقتی قطعات را نگاه می کنیم قابل درک هستند. هرچه قطعات ساده تر باشند ، احتمال اینکه هر شخص خاصی آنها را درک کند ، بیشتر است. این هنگامی مهم است که کد خود را به افراد دیگر تحویل می دهید یا برای چند ماه از کد خود دور می شوید و باید دوباره برگردید و آنچه را که انجام داده اید یاد بگیرید.

یک قیاس معماری

تصور کنید که در حال ساخت یک سازه فولادی به طول 30 فوت هستید. می توانید آن را از دسته تیرهای کوچک درست کنید که قطعات ساده ای هستند. یا می‌توانید سه قطعه بزرگ و پیچیده از فولاد بردارید و آن‌ها را کنار هم قرار دهید.

با رویکرد تیرآهن، ساخت یا خرید قطعات جداگانه آسان است. و اگر یکی خراب شود ، شما فقط آن را با یک لوازم یدکی یکسان جایگزین می کنید. ساخت ساده و نگهداری نیز ساده است.

از طرف دیگر، سه قطعه بزرگ باید با دقت ساخته شوند و بطور گسترده کار کنند. هر قطعه کامل به قدری بزرگ است که پیدا کردن و رفع همه نقص‌های آن دشوار است. و اگر بعد از اتمام ساختمان، شما عیب‌های زیادی را در هر قطعه پیدا می‌کنید، نمی‌توانید آن‌ها را جایگزین کنید - ساختمان به صورتی می‌افتد که شما یک قطعه را بیرون ببرید. بنابراین باید روی تکه‌های زشتی از فلز جوش بزنی و امیدوار باشی که همه چیز باقی بماند.

نرم افزار بسیار مشابه است - وقتی کد خود را در قطعات ساده و جدا بنویسید ، رفع نقص و نگهداری سیستم آسان است. زمانی که تکه‌های بزرگ و پیچیده را طراحی می‌کنید، هر قطعه کار زیادی لازم دارد و آنطور که باید به اندازه کافی جلا ندارد. نگهداری این سیستم دشوار می‌شود و باید به طور مداوم به آن‌ها اضافه شود تا آن را در حال دویدن نگه دارند.

پس چرا بعضی اوقات مردم به جای قطعات ساده و کوچک، نرم افزار را در قطعات بزرگ و پیچیده می نویسند؟ خوب، در هنگام ایجاد اولین نرم افزار، با صرفه جویی در وقت قابل درک است. با یک دسته از قطعات کوچک، زمان زیادی صرف کنار هم قرار دادن آنها می شود. شما نمی بینید که با قطعات عظیم - تعداد کمی از آنها وجود دارد ، آنها به هم میچسبند و تمام.

با این حال، کیفیت سیستم قطعات بزرگ بسیار پایین‌تر است و زمان زیادی صرف تعمیر آن در آینده خواهید کرد. حفظ آن سخت‌تر و سخت‌تر می‌شود در حالی که سیستم ساده آسان‌تر و آسان‌تر می‌شود. در طولانی مدت ، سادگی کارآمد است و نه پیچیدگی آن.

بنابراین ما چگونه از این قانون در دنیای عملی برنامه‌نویسی استفاده می‌کنیم؟ این موضوع بقیه این کتاب است. با این حال، به طور کلی، ایده این است که اجزای فردی کد خود را تا جای ممکن ساده کنید، و سپس اطمینان حاصل کنید که آن‌ها در طول زمان باقی بمانند. یک راه خوب برای انجام این کار، استفاده از توسعه افزایشی و روش طراحی معرفی‌شده در پایان فصل ۵ است.

از آنجا که یک گام "طراحی مجدد" قبل از اضافه کردن هر ویژگی جدید وجود دارد، می‌توانید از آن زمان برای ساده‌سازی سیستم استفاده کنید. از آنجا که قبل از اضافه شدن هر ویژگی جدید یک مرحله "طراحی مجدد" وجود دارد ، می توانید از این زمان برای ساده سازی سیستم استفاده کنید. اگرچه حتی اگر از آن روش استفاده نمی کنید، می توانید بین افزودن ویژگی ها کمی زمان بگذارید تا قطعاتی را که برای شما یا دیگر توسعه دهندگان شما خیلی پیچیده به نظر می رسند ، ساده کنید.

به این ترتیب یا روش دیگر ، شما اغلب مجبورید آنچه را که ایجاد کرده اید بگیرید و آن را ساده تر کنید - شما نمی توانید به اینکه طراحی اولیه شما همیشه مناسب است اعتماد کنید. با به وجود آمدن شرایط و نیازهای جدید ، باید قطعات سیستم را به طور مداوم از نو طراحی کنید.

به این ترتیب یا روش دیگر ، شما اغلب مجبورید آنچه را که ایجاد کرده اید بگیرید و آن را ساده تر کنید - شما نمی توانید به اینکه طراحی اولیه شما همیشه مناسب است اعتماد کنید. با به وجود آمدن شرایط و نیازهای جدید ، باید قطعات سیستم را به طور مداوم از نو طراحی کنید. مسلم است که این می تواند یک کار نسبتاً دشوار باشد. همیشه ابزار ساده ای برای نوشتن برنامه های خود به شما داده نمی شود - زبانها پیچیده هستند ، رایانه خود پیچیده است و غیره اما برای داشتن سادگی با داشته های خود تلاش کنید.

سادگی و معادله طراحی نرم افزار

شما ممکن است این را درک کرده باشید، اما این قانون مهم‌ترین چیزی را که ما می‌توانیم در حال حاضر انجام دهیم به ما می‌گوید که تلاش برای حفظ در معادله طراحی نرم‌افزار را کاهش می‌دهد. کدها را ساده‌تر کنید. برای این کار لازم نیست آینده را پیش بینی کنیم. ما فقط می توانیم به کد خود نگاهی بیندازیم، اینکه آیا کد آن پیچیده است یا خیر، و اکنون آن را برای خودمان ساده ترمی کنیم. اینگونه تلاش شما برای نگهداری با گذشت زمان کاهش می یابد - شما به طور مداوم سعی می کنید تا کدهای خود را ساده تر کنید.

مقدار مشخصی از کار در این ساده‌سازی وجود دارد، اما ایجاد تغییرات در یک سیستم ساده نسبت به سیستم پیچیده بسیار ساده‌تر است.-بنابراین اکنون کمی وقت صرف صرف ساده سازی می کنید تا بعداً در وقت خود صرفه جویی کنید.

هنگامی که تلاش برای حفظ سیستم خود را کاهش می‌دهید، مطلوبیت تمام تغییرات احتمالی را افزایش می‌دهید. (اگر می خواهید جزئیات را دوباره مرور کنید، به فصل 4 برگردید و نگاهی دوباره به معادله طراحی نرم افزار بیندازید.) ساده کردن کد شما تلاش برای نگهداری را کاهش می دهد، در نتیجه مطلوبیت هر تغییر ممکن دیگر افزایش می یابد.

سادگی نسبی است

بسیار خوب، پس ما می‌خواهیم که چیزها ساده باشند. با این حال، نحوه تعریف "ساده" واقعا به مخاطب هدف شما بستگی دارد. آنچه ساده است ممکن است برای همکارانتان ساده نباشد. هم چنین وقتی چیزی را ایجاد می‌کنید، ممکن است نسبتا "ساده" به نظر برسد، زیرا شما آن را در داخل و خارج درک می‌کنید. اما برای کسی که قبلا آن را ندیده باشد، ممکن است بسیار پیچیده به نظر برسد.

اگر می‌خواهید نقطه‌نظر کسی را درک کنید که چیزی در مورد کد شما نمی‌داند، برخی کدها را پیدا کنید که هرگز نخوانده اید و آن ها را بخوانید. سعی کنید نه تنها خطوط فردی را درک کنید، بلکه آنچه که کل برنامه انجام می‌دهد و اینکه چگونه آن را اصلاح خواهید کرد اگر مجبور بودید آن را اصلاح کنید. سعی کنید نه تنها خطوط منفرد، بلکه کار کل برنامه را بفهمید و اگر مجبور باشید چگونه آن را اصلاح می کنید. این همان تجربه ای است که دیگران هنگام خواندن کد شما تجربه می کنند. ممکن است متوجه شوید که قبل از اینکه با خواندن کد دیگران ناامید کننده باشد، سطح پیچیدگی خیلی بالا نمی رود.

به همین دلیل خوب است که بخشهایی در اسناد کد خود مانند "آیا تازه وارد این کد هستید؟" باشد که شامل برخی توضیحات ساده است که به مردم در درک کد شما کمک می کند. این موارد باید به گونه ای نوشته شود که گویی خواننده از برنامه چیزی نمی داند زیرا اگر افراد در کاری تازه کار هستند، احتمالاً چیزی در مورد آن نمی دانند.

بسیاری از پروژه های نرم افزاری این مسئله را بهم ریخته اند. شما می خواهید اسنادی را که برای توسعه دهندگان نوشته شده است بخوانید و تعداد زیادی لینک و بدون جهت به شما ارائه می شود. این برای توسعه دهنده طولانی مدت پروژه ساده به نظر می رسد ، زیرا صفحه ای با لینک های زیاد اجازه می دهد تا توسعه دهنده به سرعت به بخشی که می خواهد برود.

اما برای کسی که تازه وارد این پروژه شده ، کار پیچیده ای است. از طرف دیگر ، برای توسعه دهنده طولانی مدت ، افزودن صفحه ای با دکمه های بزرگ و ساده و حذف آن لیست پیوندها بر پیچیدگی وظیفه او می افزاید ، زیرا هدف اصلی او یافتن یک چیز بسیار خاص بسیار سریع است در اسناد. اما برای کسی که تازه وارد در پروژه هست، کار پیچیده ای است. از طرف دیگر، برای توسعه دهنده طولانی مدت، افزودن صفحه ای با دکمه های بزرگ و ساده و حذف آن لیست لینک ها بر پیچیدگی وظیفه او می افزاید ، زیرا هدف اصلی او یافتن یک چیز بسیار خاص بسیار سریع است در اسناد.

تنها چیزی که از اسناد پیچیده بدتر است، عدم وجود اسناد و مدارک است، جایی که فقط انتظار می رود آن را برای خود بنویسید یا نحوه کار کد را "قبلاً می دانید". از نظر توسعه دهنده، نحوه کار برنامه اش واضح است، اما برای دیگران کاملاً ناشناخته است.

زمینه نیز مهم است. به عنوان مثال، در متن کد برنامه ، اگر درست استفاده شود ، فناوری های پیشرفته اغلب به سادگی منجر می شوند. اما تصور کنید اگر ساختار داخلی پیشرفته چنین برنامه ای مستقیماً در یک صفحه وب به عنوان تنها رابط برنامه نمایش داده شود - در این زمینه حتی برای توسعه دهنده ساده نخواهد بود!

گاهی اوقات آنچه در یک زمینه پیچیده به نظر می رسد در زمینه ای دیگر ساده است. نمایش متن توضیحی زیادی روی بیلبورد کنار جاده بسیار پیچیده است - فقط زمانی برای رانندگان رد نمی شود که همه متن را بخوانند، بنابراین نوشتن آن در آنجا احمقانه است. اما در یک کتابچه راهنمای برنامه کامپیوتری، شامل متن های توضیحی بسیار ساده تر از ارائه توصیف یک جمله از چیزی است. به همین دلیل است که این کتاب فقط یک فصل یک خط ندارد. واقعاً آنقدر ساده نیست که فقط چیزی بگوییم و سپس توضیح ندهیم.

با توجه به همه این دیدگاه ها و زمینه های مختلف که باید در نظر گرفت، آیا این به معنای دشوار بودن دستیابی به سادگی است؟ نه اصلا. مخاطبان هدف خاصی برای همه موارد وجود دارد، و زمینه هر کار فردی که انجام می دهید معمولاً بسیار محدود است. مشکل همیشه قابل حل است. فقط مهم است که هنگام طراحی نرم افزار خود این ملاحظات را در نظر داشته باشید، بنابراین وقتی شخصی واقعاً از آن استفاده می کند ، برای آن شخص خاص ساده باشد.

جنگ ویرایشگر

در دنیای توسعه نرم افزار بحث های زیادی در مورد اینکه بهترین ابزار برای یک شغل چیست وجود دارد. مردم ویرایشگرهای متن مختلف ، زبان برنامه نویسی مختلف، سیستم عامل متفاوت و ... را دوست دارند. شاید مشهورترین "جنگ" در توسعه نرم افزار بین کاربران دو ویرایشگر متن خاص ، vim و Emacs باشد. کاربران هر یک, بعضی اوقات ادعا کرده اند که ویرایشگر مطلوب آنها اساساً از دیگری برتر است.

در واقع، به ندرت یک ابزار بسیار برتر برای نوشتن نرم‌افزار وجود دارد؛ فقط یک ابزار وجود دارد که به نظر افراد خاص برای انجام وظیفه ساده تر است. کاربران Emacs, Emacs را ساده ترین ابزار برای نوشتن نرم افزار می دانند و کاربران vim ، vim را ساده ترین ابزار می دانند. این امر تا حدی با تفاوتهای اساسی بین افراد در رابطه با نحوه دوست داشتن آنها یا نحوه تفکر آنها ارتباط دارد. مردم به سادگی اولویت های مختلفی دارند، و هیچ درست یا اشتباهی وجود ندارد. اما تا حدی بیشتر، سادگی درک شده از یک ابزار با آشنایی ارتباط دارد - هر کسی که برای مدت طولانی از یک ابزار خاص استفاده کرده باشد، احتمالاً با آن بسیار آشنا شده است ، که از نظر آن شخص، آن را بسیار ساده تر از هر ابزار دیگری می کند. برای اینکه یک ابزار جدید به همان اندازه ساده به نظر برسد، این ابزار باید بسیار ساده باشد و ویرایشگرهای متن برنامه نویسان به ندرت ساده هستند.

کسانی که برنامه نویس نیستند احتمالاً هر دو ویرایشگر متن را فراتر از تصور پیچیده می دانند, که یک مثال دیگر از این است که چگونه سادگی نسبی است.

> ☆ابزارها می توانند مشکلاتی داشته باشند که آنها را برای انجام وظیفه نامناسب سازد یا به دلایل طراحی نرم افزار انتخاب اشتباهی باشند(به "فن آوری های بد" در صفحه 62 در فصل 8 مراجعه کنید). اما با توجه به این مشکلات، سادگی نسبی یک ابزار همان چیزی است که به یک برنامه نویس اجازه می دهد تا تعیین کند که برای یک شرایط خاص چه چیزی بهتر است.

چقدر ساده باید باشید؟

زمانی که روی یک پروژه کار می‌کنید، سوالاتی در مورد سادگی می‌تواند بوجود آید. واقعاً چقدر باید ساده باشیم؟

چقدر باید یک چیز را ساده کنیم؟ آیا به اندازه کافی ساده است؟

خوب ، البته ، سادگی نسبی است. اما حتی در این صورت ، شما هنوز هم می توانید کم و بیش به سادگی برسید.

از نظر نسبی کاربر شما ، استفاده از محصول شما می تواند سخت ، آسان باشد یا جایی بین آن باشد. به همین ترتیب، از دیدگاه یک برنامه نویس دیگر، کد شما می‌تواند نسبتا سخت یا آسان برای خواندن باشد.

بنابراین ، چقدر شما باید ساده باشید؟

صادقانه؟

اگر واقعاً میخواهید موفق شوید؟

باید در حد احمقانه ای ساده باشد.

نکته خوب در مورد این سطح از سادگی این است که ، در بیشتر موارد ، هر چیزی که توسط افراد عادی قابل استفاده است توسط نوابغ نیز قابل استفاده است. شما طیف وسیعی از کاربران احتمالی را به دست می‌آورید.

اما غالباً، مردم واقعاً نمی فهمند که چقدر باید ساده احمق و گنگ باشند تا به آن سطح برسند. بیایید به یک مثال نگاه کنیم.

وقتی در بازار هستید، نقشه هایی وجود دارد که به شما می گوید همه چیز کجاست. در بهترین نقشه های بازار، یک نقطه قرمز بزرگ وجود دارد، با کلماتی که "شما اینجا هستید" با حروف بزرگ که درست روبروی شما قرار دارید. در نقشه های ساده تر، یک مثلث کوچک زرد در وسط نقشه وجود دارد که پیدا کردن آن بسیار سخت است و در کنار آن متنی وجود دارد که توضیح می دهد "مثلث کوچک زرد به معنی" شما اینجا هستید! " این را به سردرگمی کلی تلاش برای یافتن هر چیزی در این نقشه ها اضافه کنید، و شما می توانید پنج یا شش دقیقه را صرف ایستادن در مقابل آن چیز کنید، و سعی کنید بفهمید که چگونه می توانید به مکانی که می خواهید بروید برسید. برای شخصی که نقشه را طراحی کرده است ، این کاملا منطقی به نظر می رسد. او زمان زیادی را برای طراحی آن صرف کرد ، بنابراین برای او کاملا واضح است که وخوشحال خواهد بود که چندین دقیقه به آن نگاه کند، همه چیز را در مورد آن یاد بگیرد، بفهمد و غیره. اما برای ما، افرادی که در واقع از نقشه استفاده می کنند، این یک قسمت بسیار جزئی از وجود ما است.

ما فقط می خواهیم تا حد امکان ساده باشد، تا بتوانیم سریع از آن استفاده کنیم و زندگی خود را ادامه دهیم! بسیاری از برنامه نویسان به ویژه با کد خود در این مورد بد هستند. آنها تصور مي کنند که برنامه نویسان دیگر حاضر خواهند بود که زمان زیادی را برای یادگیری کد انا صرف کنند، زیرا بالاخره نوشتن آن زمان زیادی می برد! کد برای آنها مهم است، بنابراین آیا برای همه مهم نخواهد بود؟

اکنون ، برنامه نویسان به طور کلی انسان های باهوشی هستند. اما هنوز این یک اشتباه است که فکر کنیم، "برنامه نویسان دیگر همه کارهایی را که من در اینجا انجام داده ام بدون هیچ گونه ساده سازی یا توضیح کد درک خواهند کرد."

این مسئله هوشمندی نیست - این مسئله دانش است. برنامه نویسانی که کد شما را برای اولین بار میبینند چیزی در مورد آن نمی دانند. آنها باید یاد بگیرند. هرچه یادگیری آن را برای آنها آسان تر کنید، سریعتر می خواهند آن را دریابند و استفاده از آن برای آنها آسان تر خواهد بود.

روش های زیادی وجود دارد که باعث می شود کد شما به راحتی فرا گرفته شود: مستندات ساده ، طراحی ساده ، آموزش گام به گام و غیره

اما، اگر یادگیری کد شما ساده و گنگ و احمقانه نیست، مردم با آن مشکل خواهند داشت آن‌ها به اشتباه از آن استفاده می‌کنند, اشکالاتی ایجاد میکنند و سیستم را به هم میریزند. و وقتی همه این اتفاقات می افتد، از چه کسی می خواهند در مورد آن سوال کند؟ بله شما! شما قرار است وقت خود را صرف پاسخ دادن به تمام سوالات آنها کنید.(هممم , به نظر جالب می آید، این طور نیست؟)

هیچ یک از ما دوست نداریم با ما این چنین صحبت شود یا مثل اینکه ما احمق ها با ما رفتار کنند. و گاهی اوقات این امر ما را به سمت ایجاد مواردی اندک پیچیده سوق می دهد، به گونه ای که احساس می کنیم با کاربر یا دیگر برنامه نویسان صحبت نمی کنیم. ما به برخی کلمات بزرگ اشاره می‌کنیم، کمی آنها را می‌سازیم، و مردم به هوش ما احترام می‌گذارند، اما احساس حماقت می‌کنند، زیرا آن‌ها آن را **درک نمیکنند**. آنها ممکن است فکر کنند ما هوشمندانه تر از گذشته هستیم و این نوعی تملق است. اما واقعاً آیا این به آنها کمک می کند؟

از طرف دیگر ، وقتی محصول یا کد خود را به طور احمقانه ای ساده می کنید ، به دیگران اجازه می دهید آن را درک کنند. این باعث می شود که آنها احساس هوشمندی داشته باشند، به آنها اجازه می دهد آنچه را که می خواهند انجام دهند و به هیچ وجه بر شما تاثیر منفی نمی گذارد. در واقع، مردم احتمالا شما را بیشتر تحسین خواهند کرد اگر شما چیزها را ساده و ساده کنید تا اینکه آن‌ها را پیچیده کنید.

حالا تمام خانواده شما مجبور نیستند که بتوانند کد شما را بخوانند. سادگی هنوز نسبی است و مخاطبان کد نیز سایر برنامه نویسان هستند. اما برای آن برنامه نویسان دیگر، کد شما باید بسیار ساده به نظر برسد. برای دستیابی به این سادگی می تواند از فن آوری پیشرفته ای که لازم است استفاده کند، اما در نهایت باید ساده باشد.

وقتی سوال "من چقدر باید ساده باشم؟" پیش می آید, شما همچنین می توانید از خود بپرسید، "آیا من می خواهم مردم این را درک کنند و خوشحال شوند، یا می خواهم آنها گیج و ناامید شوند؟" اگر مورد اول را انتخاب کنید، فقط یک سطح از سادگی وجود دارد که موفقیت شما را اطمینان میدهد: ساده احمقانه، گنگ.

ثابت قدم باشید

سازگاری قسمت بزرگی از سادگی است. اگر کاری را یک جا در یک مکان انجام می دهید، در هر مکان باید بتوانید این کار را انجام دهید.

اگر یک متغیر را "somethingLikeThis" نامگذاری کرده اید ، تمام متغیرهای شما باید به این ترتیب نامگذاری شوند (otherNameLikeThat ، otherVariable و غیره). اگر متغیرهایی دارید که "name_like_this" هستند ، تمام متغیرها باید کوچک باشند و زیر کلمات زیر خط داشته باشند.

کدی که سازگار نباشد، درک و خواندن آن برای یک برنامه نویس دشوارتر است.

با نگاهی به مثالی از زبان طبیعی می توانیم این مسئله را نشان دهیم. این دو جمله را با هم مقایسه کنید:

- این یک جمله عادی با کلمات عادی است که همه می توانند آن را درک کنند.
- اینیکجملهعادیباکلماتعادیاستکههمهمیتوانندآنرادرککنند.

هر دوی این جمله ها دقیقاً همان حرف را میزنند، اما خواندن جمله اول ساده تر است زیرا با چگونگی نوشتن انگلیسی به انگلیسی سازگار است. مطمئناً امکان خواندن جمله دوم وجود دارد

، اما آیا شما می خواهید یک کتاب کامل مانند آن را بخوانید؟ درست. بنابراین، آیا شما می خواهید یک برنامه کامل را که بدون هیچ ثباتی نوشته شده است بخوانید؟

در برنامه نویسی شرایطی وجود دارد که مهم نیست که شما چگونه کارها را انجام می دهید ، به شرطی که همیشه آنها را به همین روش انجام دهید. از نظر تئوری، می توانستید کد خود را به روشی پیچیده و دیوانه بنویسید، اما تا زمانی که با آن سازگار باشید ، مردم یاد میگیرند که چگونه آن را بخوانند. (البته بهتر است سازگار و ساده باشید ، اما اگر نمی توانید کاملاً ساده باشید ، حداقل ثبات داشته باشید.)

ثبات کامل همچنین در بسیاری از موارد می تواند برنامه نویسی را آسان کند. به عنوان مثال، اگر هر شی در برنامه شما دارای فیلدی به نام name باشد، می توانید یک قطعه کد ساده بنویسید که مربوط به قسمت name هر شی در کل برنامه شما باشد. اما در شی A به نام a_name و شی B به نام name_of_mine, گفته شود، برای مقابله با شی A و B باید کد ویژه ای بنویسید.

به همین ترتیب، برنامه شما باید از نظر داخلی رفتار ثابتی و مطلوبی داشته باشد. برنامه نویسی که با نحوه استفاده از یک قسمت از کد شما آشنا است باید بلافاصله با نحوه استفاده از قسمت دیگری از کد شما آشنا باشد ، زیرا رفتار هر دو قطعه به یک شکل مشابه است. به عنوان مثال ، اگر هنگام استفاده از قسمت A ، برنامه نویس مجبور است سه تابع را فراخوانی کند و سپس کدی را بنویسد، هنگام استفاده از قسمت B، او باید یك مجموعه مشابه از سه تابع را فراخوانی کند و سپس کدی را بنویسد. و اگر تابعی به نام dump در قسمت A دارید که باعث می شود قسمت A تمام متغیرهای داخلی خود را چاپ کند ، تابعی که dump در قسمت B نام دارد باید همان کار را برای قسمت B انجام دهد. هر وقت برنامه نویسان به بخش جدیدی از سیستم نگاه می کنند ، مجبور نشوید برنامه سیستم را دوباره یاد بگیرند. شاید همه چیز در دنیای واقعی چندان سازگار نباشد، اما شما مسئول دنیای برنامه خود هستید، بنابراین می توانید کارها را ساده و سازگار کنید.

چند نمونه از ثبات در دنیای واقعی وجود دارد. در بیشتر آسیا، مردم از چاپستیک های چوبی برای خوردن استفاده می کنند. در قاره آمریکا و اروپا مردم از چنگال استفاده می کنند. بسیار خوب، این دو روش مختلف غذا خوردن است، اما به طور کلی در هر منطقه مشخص کاملاً سازگار است. حال تصور کنید اگر هر وقت به خانه کسی می رفتید، مجبور بودید روش کاملاً جدیدی برای غذا خوردن یاد بگیرید. شاید در خانه ی "باب" آنها با قیچی غذا می خورند و در خانه "مری" با تکه های مقوایی صاف غذا می خورند. غذا خوردن کاملاً پیچیده می شود ، نه؟

در برنامه نویسی هم همین طور است - بدون ثبات ، همه چیز پیچیده می شود. با ثبات، آنها ساده می شوند. و حتی اگر آنها ساده نباشند، حداقل شما می توانید پیچیدگی را فقط یک بار یاد بگیرید، و سپس آن را برای همیشه می دانید.

خوانایی

همانطور که در بسیاری از زمان‌ها در جهان توسعه نرم‌افزار گفته شده‌است، کد بیشتر از آنچه نوشته می‌شود، خوانده می‌شود. پس مهم است که متن را آسان کنید:

این کد در درجه اول به نحوه اشغال فضا توسط حروف و نماد بستگی دارد.

اگر کل جهان سیاه بود، نمی توانستید اجسام را از هم جدا کنید. همه آنها یک توده سیاه بودند. به همین ترتیب، اگر یک فایل کامل توده ای از کد بدون فاصله منطقی و منطقی کافی باشد، جدا کردن قطعات دشوار است. فضا همان چیزی است که چیزها را جدا نگه می دارد.

شما فضای زیادی نمی خواهید، زیرا در این صورت تشخیص نحوه ارتباط امور دشوار است. و شما خیلی کم نمی خواهید، زیرا در این صورت تشخیص اینکه همه چیز از هم جدا است دشوار است.

هیچ قانون سخت و سریعی در مورد نحوه دقیق فاصله گذاری کد وجود ندارد ، به جز اینکه باید به روشی ثابت انجام شود و فاصله گذاری باید به خواننده در مورد ساختار کد درباره ساختار کد کمک کند.

مثال: فضاهای داخلی

خواندن این کد دشوار است زیرا فضای کمی در آن وجود دارد - اطلاعات کمی در مورد ساختار کد ارائه می شود:

```
x=1+2;y=3+4;z=x+y;if(z>y+x){print"error";}
```

اینجا همان بلوک از کد با فضای بسیار زیاد در آن است - فضا مانع از دیدن ساختار کد می‌شود:

```
x         =         1+        2;
y=3            +4;

  z = x        +    y;
If (z    >     y+x)
  {    print "error" ;
}
```

خواندن بیشتر از کد با فضای خالی سخت‌تر است. این همان کد با فاصله‌گذاری منطقی است:

```
x = 1 + 2
y = 3 + 4
z = x + y
If (z > y + x){
    Print “error”;
}
```

خواندن آن بسیار راحت تر است و به شما کمک می کند تا متوجه شوید که برنامه نویس قصد طراحی برنامه را دارد. سه متغیر تنظیم می شود ، و سپس در برخی شرایط، یک خطا ایجاد می شود. این ساختار سیستم است که با روشی که برنامه‌نویس از فضای کد استفاده می‌کند به خواننده نشان داده می‌شود.

خواندن کد به سهولت نیز در تعمیر کد کمک می کند. در مثال قبلی، هنگامی که کد به درستی فاصله دارد، به راحتی می توانیم ببینیم که z هرگز از x + y بزرگتر نخواهد بود ، زیرا z همیشه برابر با y + x است. بنابراین ، بلاک شروع شده با (z\> y + x) باید حذف شود ، زیرا ضروری نیست.

به طور کلی ، اگر کد چند حفره ای دارید که خواندن آن نیز دشوار است، اولین کاری که باید انجام دهید این است که آن را بیشتر خوانا کنید. از اینکه می توانید با وضوح بیشتری اشکالات را ببینید.

نامگذاری چیزها

بخش مهمی از خوانایی، دادن نامهای خوب به متغیرها، توابع، کلاسها و غیره است. در حالت ایده آل:

> **اسامی باید به اندازه کافی طولانی باشند تا به طور کامل هر چیزی را که هست، منتقل کنند، بدون اینکه بیش از حد طولانی باشند که خواندن آن دشوار شود.**

همچنین مهم است که در مورد چگونگی استفاده از تابع ، متغیر و غیره فکر کنید. هنگامی که شروع به قرار دادن نام آن در خطوط کد می کنیم، آیا آن خطوط کد را آنقدر طولانی می‌سازد که خواندن آن دشوار باشد؟ به عنوان مثال ، اگر تابعی دارید که فقط یک بار فراخوانی می شود، به تنهایی در یک خط (بدون هیچ کد دیگری در آن خط) می تواند یک نام نسبتاً طولانی داشته باشد. با این حال، تابعی که قرار است مرتباً در عبارت پیچیده از آن استفاده کنید، باید نام کوتاهی داشته باشد (گرچه هنوز به اندازه کافی طولانی است که به طور کامل آن چه انجام می‌دهد را منتقل کند)

توضیحات

داشتن توضیحات خوب در کد، بخش عمده اي از قابل خواندن کردن آن است. با این حال ، معمولاً نباید توضیحاتی را اضافه کنید که می گوید قطعه کد چه کاری انجام می دهد. این باید از خواندن کد مشخص باشد. اگر واضح نیست، کد باید ساده تر شود. فقط در صورتی که نمی توانید کد را ساده تر کنید، باید توضیحی درباره عملکرد آن بدهید.

هدف واقعی از توضیحات این است که چرا کاری انجام داده اید، در حالی که دلیل آن مشخص نیست. اگر این را توضیح ندهید، ممکن است برنامه نویسان دیگر گیج شوند و وقتی می خواهند کد شما را تغییر دهند ممکن است قسمت های مهم آن را حذف کنند.

برخی از افراد معتقدند که قابلیت خواندن پایان و سرانجام سادگی کد است - اگر کد شما آسان خوانده شود، تمام آنچه را که باید به عنوان یک طراح انجام دهید، انجام داده اید. این درست نیست - شما می‌توانید کدی قابل خواندن داشته باشید ولی هنوز سیستمی داشته باشید که بسیار پیچیده باشد. با این حال، خوانایی کد شما بسیار مهم است و معمولاً اولین قدم است که باید در راه طراحی نرم افزار خوب برداشته شود.

مثال : نامها

در اینجا چند کد با نام های واقعا ضعیف آورده شده است:

```
q = s(j, f, m);

p(q);
```

این نام‌ها با آنچه که متغیرها انجام می‌دهند و آنچه که توابع انجام می‌دهند، ارتباط برقرار نمی‌کنند. این همان کد با نام‌های خوب است:

```
quarterly\_total = sum(january, february, march);

print(quarterly\_total);
```

و در اینجا دوباره همان کد وجود دارد، با اسامی بسیار طولانی که خواندن آنها دشوار است:

```
quarterly_total_for_company_in_2011_as_of_today =
add_all_of_these_together_and_return_the_result(january_total_amount,
february_total_amount, march_total_amount);
send_to_screen_and_dont_wait_for_user_to_respond(quarterly_total_for_company_in_2011_as
```

این نام ها فضای زیادی را اشغال می کنند که خواندن آنها را دشوار می کند. بنابراین، به نوعی نامگذاری چیزها نیز به چگونگی اشغال فضا به حروف و نمادها بازمی گردد.

سادگی نیاز به طراحی دارد

متاسفانه ، مردم به طور طبیعی سیستم های ساده ای نمی سازند. بدون توجه به طراحی، یک سیستم به یک هیولای عظیم و پیچیده تبدیل خواهد شد.

اگر پروژه شما از طراحی خوبی برخوردار نباشد و به رشد خود ادامه دهد، در نهایت با پیچیدگی بی پایانی مواجه خواهید شد. درک این موضوع برای بعضی از مردم سخت است - برخی نمی توانند تصور کنند که آینده ای فراتر از فردا وجود دارد و برخی تجربه کافی برای درک چگونگی پیچیدگی چیزها را نداشته اند. و یک فرهنگ سازمانی می‌تواند وجود داشته باشد که می‌گوید: "اوه ، ما فقط ویژگی های جدید را به صورت نیمه کاره ارائه میدهیم, ما باید کارها را به روش صحیح انجام دهیم، اما نمی توانیم این کار را انجام دهیم." اما یک روز، پروژه شما با شکست روبرو خواهد شد. و مهم نیست که چند دلیل برای این شکست وجود داشته باشد، این واقعیت که پروژه شما شکست خورد را تغییر نخواهد داد.

از طرف دیگر، وقتی خوب طراحی کردید، اغلب اعتبار زیادی برای شما وجود ندارد. شکست‌های فاجعه‌آمیز در طراحی, بزرگ و قابل‌توجه هستند، در حالی که کارهای کوچک برای دستیابی به یک طراحی خوب برای افرادی که از نزدیک با کد در ارتباط نیستند قابل مشاهده نیست. این می تواند طراح بودن را به یک کار سخت تبدیل کند. رسیدگی به یک شکست بزرگ باعث تشکر فراوان از شما می شود، اما جلوگیری از وقوع آن ... خوب، به احتمال زیاد هیچ کس متوجه نمیشود.

بنابراین، اجازه دهید در اینجا به شما تبریک بگویم. کمی به طراحی فکر کردید؟ عالی! کاربران و توسعه دهندگان دیگر مزایای آن را مشاهده خواهند کرد - نرم‌افزار سالم، انتشار به موقع، و یک codebase واضح و قابل درک. در کار خود احساس اعتماد

به نفس خواهید کرد و احساس موفقیت خواهید کرد. آیا سایر توسعه دهندگان می دانند که چقدر کار شده است تا همه چیز به همین راحتی اجرا شود؟ شاید نه.اما مشکلی نیست. علاوه بر تبریک همتایان خود، پاداش‌های دیگری هم در دنیا وجود دارد.

به هر حال، یک‌بار در یک زمان خاص، از همه کارهای شما قدردانی خواهد شد. ناامید نشوید - در نهایت کسی متوجه خواهد شد. و تا آن زمان ، از تمام نتایج مثبت طراحی موثر و درست لذت ببرید.

☆هنگامی که شما شروع به استفاده از اصول طراحی در این کتاب برای پروژه خود می کنید، ممکن است مدتها طول بکشد تا برخی از برنامه نویسان یا همکاران ارشد شما درک کنند که چرا آنها نیز باید خوب طراحی کنند. خواندن این کتاب به آنها کمک خواهد کرد. اگر آنها نمی توانند یا نمی خواهند آن را بخوانند ، آنها را به سمت تصمیمات خوب طراحی هدایت کنید (یا در بدترین حالت آنها را مجبور کنید) ، و بعد از چند سال(در خارج از پروژه) می فهمند که تصمیمات خوب طراحی چگونه نتیجه می دهند.

فصل 9

پیچیدگی

هنگامی که شما به عنوان یک برنامه نویس حرفه ای کار می کنید، این احتمال وجود دارد که کسی را بشناسید (یا کسی هستید!) که این داستان ترسناک مشترک را پشت سر میگذارد : "ما پنج سال پیش کار بر روی این پروژه را شروع کردیم ، و فناوری که ما از آن استفاده می کردیم / می ساختیم آن زمان مدرن بود، اما اکنون منسوخ شده است. کارها با این فناوری منسوخ پیچیده تر و پیچیده تر می شوند، بنابراین احتمال اینکه من پروژه را تمام کنم کمتر و کمتر می شود. اما اگر بازنویسی کنیم، می توانیم پنج سال دیگر اینجا باشیم! "

یکی دیگر از داستان های معروف این است: "ما نمی توانیم به اندازه کافی سریع پیشرفت کنیم تا بتوانیم مطابق با نیازهای مدرن کاربر باشیم." یا "در حالی که ما در حال توسعه بودیم، شرکت X سریعتر از ما, محصولی بهتر از محصول ما نوشت." اکنون می دانیم که منشأ این مشکلات *پیچیدگی* است. شما با یک پروژه ساده شروع می کنید که می تواند در یک ماه به پایان برسد. سپس پیچیدگی اضافه می کنید و کار سه ماه طول می کشد. سپس هر قطعه از آن را می گیرید و پیچیده تر می کنید، و کار نه ماه طول می کشد

پیچیدگی بر پیچیدگی می‌افزاید - این فقط یک امر خطی نیست. یعنی، شما نمی‌توانید فرضیاتی مانند این داشته باشید: "ما 10 ویژگی داریم، بنابراین با افزودن 1 ویژگی دیگر فقط 10 درصد زمان بیشتر اضافه می شود." در واقع، این یک ویژگی جدید باید با تمام 10 ویژگی موجود شما هماهنگ شود. بنابراین، اگر ۱۰ ساعت زمان کد نویسی برای پیاده‌سازی این ویژگی طول بکشد، ممکن است ۱۰ ساعت دیگر طول بکشد تا ۱۰ ویژگی موجود به درستی با ویژگی جدید تعامل داشته باشند. هرچه ویژگی ها بیشتر باشد، هزینه افزودن یک ویژگی بالاتر می رود. با داشتن یک طرح نرم‌افزاری عالی می‌توانید این مشکل را به حداقل برسانید، اما با این وجود همیشه هزینه های اضافی جزئی برای هر ویژگی جدید وجود دارد.

برخی از پروژه ها با چنان مجموعه پیچیده ای از الزامات شروع می شوند که هرگز نسخه اول را منتشر نمیکنند. اگر در این وضعیت هستید، باید فقط ویژگی‌ها را اصلاح کنید. در اولین انتشار به خود سخت نگیرید— چیزی را منتشر کنید که کار بکند و با گذشت زمان بر کارایی آن بیافزایید.

راه‌های دیگری به غیر از افزودن ویژگی‌ها برای اضافه کردن پیچیدگی وجود دارد. معمول‌ترین روش‌ها عبارتند از:

گسترش هدف نرم افزار

به طور کلی، فقط هرگز این کار را نکنید. بخش بازاریابی شما ممکن است از فکر ساخت یک نرم افزار واحد که مالیات چندین نیاز را همزمان برطرف میکند، بسیار خوشحال باشد، اما هر زمان که پیشنهادی از این دست به میز کار شما می آید، باید تا جایی که می توانید بلند فریاد بزنید. به هدف موجود نرم افزار خود پایبند باشید. فقط باید کاری را انجام دهد که میتواند به خوبی انجام دهد، و شما موفق خواهید شد. (به شرطی که نرم افزار شما به افراد در مورد چیزی که واقعاً به آن نیاز دارند کمک کند)

افزودن برنامه نویسان

بله، درست است - افزودن افراد بیشتر به تیم ، کارها را ساده تر نمی کند. در عوض ، این پیچیدگی را اضافه می کند. یک کتاب معروف به نام "The Mythical ManMonth" توسط فرد بروکس نوشته شده است که این موضوع را مشخص می‌کند. اگر 10 برنامه نویس دارید، افزودن یازدهمین برنامه نویس به معنای صرف وقت برای ایجاد شیار بین برنامه نویس جدید و برنامه نویسان دیگر است ، به علاوه زمان صرف شده توسط شخص جدید در تعامل با 10 برنامه نویس موجود ، و غیره. احتمال موفقیت شما در گروه کوچکی از برنامه نویسان خبره بیشتر از گروه بزرگی از برنامه نویسان غیر متخصص است.

تغییر چیزهایی که لازم نیست تغییر کنند

هر زمان چیزی را تغییر دهید، پیچیدگی اضافه می کنید. چه این مورد نیاز باشد، طراحی، یا فقط یک قطعه کد، شما امکان ایجاد تغییر را، زمان مورد نیاز برای اجرای تغییر، زمان مورد نیاز برای ارزیابی تغییر، زمان مورد نیاز برای پی‌گیری تغییر، و زمان مورد نیاز برای آزمودن تغییر، را در نظر بگیرید. هر تغییر براساس تمام این پیچیدگی ساخته می‌شود، بنابراین هر چه بیشتر تغییر کنید، زمان بیشتری برای هر تغییر جدید ایجاد می‌شود. هنوز هم مهم است که تغییرات خاصی ایجاد کنید، اما باید تصمیمات آگاهانه در مورد آن‌ها اتخاذ کنید، نه فقط تغییر برای هوس.

محبوس بودن در تکنولوژی‌های بد

اساسا، این جایی است که شما تصمیم می‌گیرید از برخی از فن‌آوری ها استفاده کنید، و سپس برای مدتی طولانی با آن درگیر می‌شوید چون شما به آن وابسته هستید. یک فناوری به این معنا اگر شما را محصور کند "بد" است (به شما اجازه نمی دهد در آینده به راحتی به فناوری دیگری بروید) ، به اندازه کافی برای نیازهای آینده انعطاف پذیر نخواهد بود یا فقط از سطح کیفی که برای طراحی نرم افزار ساده با آن نیاز دارید را داشته باشید.

سوء تفاهم

برنامه نویسان که کاملاً کار خود را درک نمی کنند ، تمایل به توسعه سیستم های پیچیده دارند. این می‌تواند یک چرخه معیوب باشد: سو تفاهم منجر به پیچیدگی می‌شود که منجر به سو تفاهم بیشتر و غیره می‌شود. یکی از بهترین روش‌ها برای بهبود مهارت‌های طراحی شما این است که مطمئن شوید که سیستم و ابزارهایی که با آن‌ها کار می‌کنید را به طور کامل درک می‌کنید. هرچه بیشتر این موارد را درک کنید، و هر چه بیشتر در مورد نرم‌افزار بدانید، طرح‌های شما ساده تر خواهند بود.

طراحی ضعیف یا بدون طراحی

اساساً، این فقط به معنای "عدم برنامه ریزی برای تغییر" است. اوضاع در حال تغییر است و برای حفظ سادگی در حین رشد پروژه کار طراحی لازم است. شما باید در ابتدا خوب طراحی کنید و همچنین با گسترش سیستم به طراحی خود ادامه دهید - در غیر این صورت ، می توانید پیچیدگی عظیم را خیلی سریع بوجود بیاورید، زیرا با یک طراحی ضعیف، هر ویژگی جدید پیچیدگی کد را چند برابر می کند به جای اینکه فقط کمی پیچیدگی اضافه کنید.

اختراع دوباره چرخ

اگر به عنوان مثال، پروتکل خود را در صورت وجود پروتکل کاملاً خوب دیگر دوباره اختراع کنید، وقت زیادی را صرف کار روی پروتکل خواهید کرد ، زمانی که می توانید روی نرم افزار خود کار کنید. شما تقریبا هرگز نباید وابستگی زیادی اختراع دوباره چیزی داشته باشید، مانند یک سرور وب، یک پروتکل یا یک کتابخانه بزرگ، مگر این که محصول شما باشد. تنها زمان‌هایی که برای دوباره اختراع چرخ مناسب است، زمانی است که هر یک از موارد زیر درست هستند:

- شما به چیزی احتیاج دارید که هنوز وجود ندارد.

- تمام "چرخ" های موجود فناوری های بدی هستند که شما را قفل می کنند.
- "چرخ" های موجود اساساً قادر به تأمین نیازهای شما نیستند.
- "چرخ" های موجود به درستی نگهداری نمی شوند و شما نمی توانید تعمیر و نگهداری آنها را به عهده بگیرید (به عنوان مثال ، کد منبع ندارید).

همه این عوامل به آرامی و به تدریج برای پروژه شما مضر هستند، بلافاصله مخرب نیستند. بیشتر آنها فقط آسیب های طولانی مدت میزنند - چیزی که یک سال یا بیشتر نمی بینید - بنابراین وقتی کسی آنها را پیشنهاد می کند، اغلب بی خطر به نظر می رسند. و حتی وقتی که شروع به اجرای آن‌ها می‌کنید، ممکن است خوب به نظر برسند. اما با گذشت زمان - و به خصوص بیشتر و بیشتر این انباشت - پیچیدگی آشکارتر می‌شود و رشد می‌کند و رشد می‌کند، تا زمانی که شما یک قربانی دیگر از آن داستان وحشتناک میشوید.

پیچیدگی و هدف

هدف اصلی هر سیستم معینی که شما روی آن کار می کنید باید کاملاً ساده باشد. این امر کمک می کند تا سیستم در کل تا جایی که میتواند واقع بینانه ای ساده باشد. اما اگر شروع به اضافه کردن ویژگی هایی کنید که هدف دیگری را برآورده میکنند، همه چیز خیلی زود پیچیده می شوند. برای مثال، هدف اولیه یک پردازشگر کلمه، کمک به شما برای نوشتن چیزی است. اگر ما به طور ناگهانی آن را قادر به خواندن ایمیل شما کنیم، به طرز مضحکی پیچیده می‌شد. می‌توانید تصور کنید که رابط کاربر چه شکلی خواهد بود؟ همه دکمه ها را کجا قرار می دهید؟ ما می‌گوییم که این هدف پردازشگر کلمه شما نقض میکند. شما حتی هدف آن را گسترش ندادید. شما فقط ویژگی هایی را اضافه کردید که هیچ ارتباطی با آن ندارند.

همچنین مهم است که به هدف کاربر فکر کنید. کاربر شما در تلاش است کاری انجام دهد. در حالت ایده آل ، هدف یک برنامه باید بسیار نزدیک به هدف کاربر باشد. برای مثال ، بگذارید بگوییم هدف کاربر انجام مالیات خود است. او نرم افزاری می خواهد که هدف آن کمک به مردم در پرداخت مالیات خود باشد.

اگر هدف شما و کاربر با هم مطابقت نداشته باشد، احتمالاً زندگی او را دشوار می کنید. به عنوان مثال، اگر او می خواهد ایمیل خود را بخواند، اما هدف اصلی برنامه ای که استفاده می کند نمایش تبلیغات به کاربران است، این اهداف مطابقت ندارند.

آیا می خواهید کاربر شما خیلی سریع عصبانی شود؟ تحقق هدفش را برای او دشوار کنید. هنگام تلاش برای انجام کاری، پنجره های نامربوط را باز کنید. آنقدر ویژگی به برنامه خود اضافه کنید که او نتواند ویژگی مناسب را پیدا کند. از نمادهای عجیب و غریب زیادی استفاده کنید که او نمی فهمد. روش های زیادی برای انجام این کار وجود دارد ، اما همه آنها منجر به تداخل در هدف کاربر یا نقض هدف اصلی خود برنامه می شود.

بعضی اوقات ، بازاریابان یا مدیران برای برنامه ای اهدافی دارند که در واقع با هدف اصلی برنامه مطابقت ندارد ، مانند "زیبا باشد " " طراحی بروز داشته باشد" "محبوب رسانه های خبری شود" "از آخرین فناوری ها استفاده شود" و غیره.

این افراد ممکن است برای سازمان شما مهم باشند، اما افرادی نیستند که باید تصمیم بگیرند که برنامه شما چه کاری انجام می دهد! به عنوان یک طراح نرم افزار یا مدیر فنی، وظیفه شما این است که ببینید برنامه در مسیر خود قرار دارد و هرگز هدف اصلی خود را نقض نمی کند. هیچ کس دیگری مسئولیت آن را بر عهده نخواهد گرفت. گاهی اوقات ممکن است واقعاً مجبور شوید برای آن بجنگید، اما در طولانی مدت ارزشش را دارد.

و اینطور نیست که شما با این فلسفه به یک شکست بازاریابی رسیده‌اید. محصولات بسیار بسیار زیادی وجود دارند که تنها با رعایت یک هدف ، بسیار موفق بوده اند. هدف صابون فقط تمیز کردن است. هدف نمک شور کردن است .یک لامپ فقط اتاق

را روشن می کند. اما همه اینها محصولاتی هستند که طی دهه ها از شرکت های عظیم حمایت کرده اند. برای داشتن بازاریابی موثر نیازی به داشتن محصول پیچیده نیست. شما فقط باید در بازاریابی دانش و مهارت داشته باشید که یک زمینه کاملاً جدا از طراحی نرم افزار است. واقعاً ، نیازی به تجملات و پیچیدگی و سعی برای انجام 500 کار همزمان در یک برنامه وجود ندارد. کاربران با یک محصول متمرکز و ساده که هرگز هدف اساسی خود را نقض نمی کند خوشحال هستند.

فناوری های بد

یکی دیگر از منابع متداول پیچیدگی ، انتخاب فناوری نادرست برای استفاده در سیستم شماست - خصوصاً یکی از فناوری هایی که در نهایت نمی تواند نیازهای آینده را تحمل کند. با این حال ، بدون اینکه بتوانید آینده را پیش بینی کنید ، اکنون چه فناوری را انتخاب کنید ، می تواند مشکل باشد. خوشبختانه ، سه عامل وجود دارد که می توانید برای تعیین "بد" بودن فناوری قبل از شروع استفاده از آن بررسی کنید: پتانسیل بقا، قابلیت همکاری و توجه به کیفیت.

پتانسیل بقا

پتانسیل بقای یک فناوری احتمال ادامه کار آن است. اگر در یک کتابخانه منسوخ شده باشید، واقعاً دچار مشکل خواهید شد.

با نگاهی به تاریخچه انتشار اخیر آن، می توانید از پتانسیل زنده ماندن یک نرم افزار مطلع شوید. آیا توسعه دهندگان مرتباً با نسخه های جدیدی که مشکلات واقعی کاربران را برطرف می کند ، ارائه میکنند؟ همچنین ، توسعه دهندگان چقدر به گزارشات اشکالات پاسخ می دهند؟ آیا آنها ایمیل لیست یا تیم پشتیبانی بسیار فعال دارند؟ آیا افراد آنلاین زیادی در مورد این فناوری صحبت می کنند؟ اگر اکنون یک فناوری تحرک زیادی دارد ، می توانید مطمئن باشید که به زودی نمی میرد.

همچنین بررسی کنید که آیا فقط یک فروشنده از آن فن آوری پشتیبانی میکند یا اینکه به طور گسترده ای در بسیاری از زمینه های نرم افزار توسط بسیاری از توسعه دهندگان مختلف پذیرفته شده است. اگر فقط یک فروشنده وجود داشته باشد که سیستم را پشتیبانی میکند و به جلو می فرستد، این خطر وجود دارد که آن فروشنده یا از کار خارج شود یا فقط تصمیم به حفظ و ادامه سیستم بگیرد.

محبوبیت

ممکن است به نظر برسد که ما می گوییم شما فقط باید محبوب ترین فناوری را متناسب با نیازهای خود انتخاب کنید. تا حدی ، این درست است - فن آوری های محبوب دارای پتانسیل زنده ماندن زیادی هستند. با این حال ، شما باید تفاوت بین ابزارهای معتبر محبوب و ابزارهای محبوب فقط به دلیل داشتن نوعی انحصار را بررسی کنید.

در زمان نگارش این کتاب، C یکی از نمونه های زبان مشهور معتبری است. بسیاری از افراد از آن در سازمانهای مختلف برای اهداف مختلف استفاده می کنند. این موضوع چندین استاندارد بین المللی است، و تعداد زیادی پیاده سازی از این استانداردها وجود دارد، از جمله کامپایلرهای بسیار پرکاربرد مختلف.

برخی از فناوری ها فقط به این دلیل محبوب هستند که شما باید از آنها استفاده کنید.فرض کنید شرکت X زبان برنامه نویسی خود را طراحی کند. سپس یک دستگاه محبوب را طراحی می کند که فقط برنامه های نوشته شده به آن زبان را می پذیرد. این مورد "یک فروشنده" ذکر شده در متن است - این زبان ممکن است محبوب به نظر برسد ، اما در واقع پتانسیل زنده ماندن ضعیفی دارد ، مگر اینکه به طور گسترده در صنعت نرم افزار مورد استفاده قرار گیرد.

قابلیت همکاری

قابلیت همکاری اندازه گیری این است که در صورت لزوم جدا شدن از یک فناوری آسان است. برای اینکه تصوری از قابلیت همکاری یک فناوری داشته باشید، از خود بپرسید ، "آیا می توانیم به روشی استاندارد با این فناوری تعامل داشته باشیم ، بنابراین تغییر سیستم به سیستم دیگری که از همان استاندارد پیروی می کند آسان خواهد بود؟"

به عنوان مثال ، استانداردهای بین المللی برای چگونگی تعامل یک برنامه با سیستم پایگاه داده وجود دارد. برخی از سیستم های پایگاه داده به خوبی از این استانداردها پشتیبانی می کنند. اگر یکی از این سیستم های پایگاه داده خوب را انتخاب کنید ، می توانید در آینده فقط با تغییرات جزئی در برنامه خود به سیستم پایگاه داده دیگری بروید.

با این حال، برخی دیگر از سیستم های پایگاه داده در پشتیبانی از استانداردها مهارت چندانی ندارند. اگر می خواهید بین سیستم های پایگاه داده ای که از استاندارد پشتیبانی نمی کنند جابجا شوید، باید برنامه خود را دوباره بنویسید. بنابراین ، وقتی یکی از این سیستم های غیراستاندارد را انتخاب می کنید ، در آن گیر کرده اید و دیگر نمی توانید به راحتی به سیستم دیگری بروید.

توجه به کیفیت

این یکی بیشتر یک اندازه گیری ذهنی است ، اما ایده این است که ببینیم آیا محصول در عرضه های اخیر بهتر شده است یا خیر. اگر می توانید کد منبع را ببینید، بررسی کنید که آیا توسعه دهندگان در حال تجزیه و تحلیل و تمیز کردن کدبیس هستند. آیا استفاده از آن آسانتر شده یا پیچیده تر؟ آیا افرادی که این فناوری را حفظ می کنند در واقع به کیفیت محصول خود اهمیت می دهند؟

دلایل دیگر

جنبه های دیگری نیز وجود دارد که باید هنگام انتخاب یک فناوری در نظر بگیرید - در درجه اول سادگی و مناسب بودن آن برای اهداف شما. نظر شخصی نیز می تواند نقشی داشته باشد، بعد از اینکه تمام ملاحظات عملی را در نظر گرفتید. بعضی از افراد شکل ظاهری یک زبان برنامه نویسی را بهتر از زبان دیگر دوست دارند. این گاهی اوقات می تواند یک دلیل معتبر برای انتخاب یک فناوری باشد - اگر شما فقط یک فناوری را بیشتر از فناوری دیگر دوست دارید و هر چیز دیگری بین آنها برابر است، با یکی از ویژگی هایی که شما را خوشحال می کند تکنولوژی خود را انتخاب کنید. از این گذشته، شما کسی هستید که از آن استفاده خواهید کرد - نظر شما مهم است! دستورالعمل های بالا به شما کمک می کند تا گزینه های قطعاً بد را پاک کنید. بقیه به تحقیقات شخصی، نیازها و خواسته های شما بستگی دارد.

پیچیدگی و راه حل اشتباه

غالباً ، اگر مشکلی بسیار پیچیده باشد، این بدان معنی است که در طراحی جایی در زیر سطح پیچیدگی، خطایی رخ داده است.

به عنوان مثال، ساخت یک ماشین با سرعت بسیار دشوار است اگر چرخ‌های مربعی داشته باشند. تنظیم موتور مشکلی را حل نمی کند —شما باید ماشین را دوباره طراحی کنید تا چرخ های آن گرد باشد.

هر زمان که در برنامه شما "پیچیدگی غیرقابل حل" وجود داشته باشد، به این دلیل است که اساساً اشکالی در طراحی وجود دارد. اگر مشکل در یک سطح غیر قابل حل به نظر میرسد، به عقب برگردید ببینید چه چیزی ممکن است زمینه ساز این مشکل باشد. برنامه نویسان اغلب این کار را انجام می دهند. ممکن است خودتان را در وضعیتی پیدا کنید که بگویید: "من این کد بسیار نامرتب را دارم و افزودن ویژگی جدید واقعاً پیچیده است!" خوب، مشکل اصلی شما این است که این کد به هم ریخته‌است. آن را مرتب کنید، کد موجود را ساده کنید، و خواهید دید که اضافه کردن ویژگی‌های جدید نیز ساده خواهد بود.

سعی در حل چه مشکلی دارید؟

اگر کسی به شما مراجعه کند و چیزی مانند این را بگوید ، "چگونه با یک راکت به ماه بروم ؟" سوالی که باید بپرسید این است که "چه مشکلی دارید که سعی دارید آن را حل کنید؟" ممکن است متوجه شوید آنچه که این شخص واقعاً به آن نیاز دارد جمع آوری برخی از سنگهای خاکستری است. چرا فکر می‌کرد باید به ماه پرواز کند و از یه راکت برای انجام این کار استفاده کند. فقط او ممکن است بداند. مردم اینگونه گیج می شوند. با این حال از آنها بپرسید که برای حل چه مشکلی تلاش می کنند و یک راه حل ساده خود را نشان می دهد. به عنوان مثال ، در این حالت ، هرگاه مسئله را کاملاً فهمیدیم ، راه حل ساده و آشکار می شود: او باید فقط بیرون برود و برخی از سنگهای خاکستری را پیدا کند - نیازی به یک راکت نیست.

بنابراین ، وقتی همه چیز پیچیده شد ، به عقب برگردید و به مشکلی که می خواهید حل کنید، نگاهی بیندازید. یک قدم واقعاً بزرگ به عقب بردارید. شما مجاز هستید همه چیز را زیر سوال ببرید. شاید شما فکر می کردید که اضافه کردن دو و دو, تنها راه برای بدست آوردن چهار است، و به این فکر نکرده اید که یک و سه را اضافه کنید، به طور کامل از اضافه کردن صرف نظر کنید و فقط چهار را در آنجا قرار دهید. مسئله این است، "چگونه شماره چهار را بدست آورم؟"

هر روشی برای حل این مسئله قابل قبول است ، بنابراین آنچه شما باید انجام دهید این است که بفهمید بهترین روش برای شرایطی که در آن قرار دارید چیست.

فرضیات خود را کنار بگذارید. واقعاً به مشکلی که می خواهید حل کنید، نگاه کنید. اطمینان حاصل کنید که تمام جنبه های آن را کاملاً درک کرده اید و سپس ساده ترین راه حل آن را رقم بزنید. نپرسید، "چگونه می توانم با استفاده از کد فعلی خود این مشکل را حل کنم؟" یا "چگونه پروفسور این مشکل را در برنامه خود حل کرد؟" نه — فقط از خود بپرسید، "به طور کلی، در یك جهان کامل، چگونه باید این نوع مشکلات حل شود؟" از آنجا می‌توانید ببینید که چگونه کد شما باید بازسازی شود. سپس می توانید کد خود را دوباره بنویسید. و بعد از آن می توانید مشکل را حل کنید.

مشکلات پیچیده

گاهی اوقات از شما خواسته می شود مشکلی را حل کنید که ذاتاً بسیار پیچیده است — به عنوان مثال ، اصلاح هجی کردن، یا کامپیوتری را مجبور کنیم که شطرنج بازی کند. این بدان معنا نیست که راه حل شما باید پیچیده باشد، اما به این معنی است که شما باید هنگام کار با این مشکل بیش از حد معمول کار کنید تا کد خود را ساده کنید.

اگر با یک مشکل پیچیده سروکار دارید، آن را روی کاغذ به زبان ساده بنویسید یا آن را به عنوان نمودار بکشید. بعضی از بهترین برنامه نویسی ها در واقع روی کاغذ انجام می‌شود. قرار دادن آن در کامپیوتر فقط جزئیات کوچک است.

اکثر مسائل دشوار طراحی را میتوان به ساده با کشیدن و یا نوشتن آن‌ها روی کاغذ حل کرد.

مدیریت پیچیدگی

به عنوان یک برنامه نویس، با پیچیدگی روبرو خواهید شد. برنامه نویسان دیگر برنامه های پیچیده ای را می نویسند که باید آنها را برطرف کنید. طراحان سخت افزار و طراحان زبان زندگی شما را دشوار می کنند. اگر بخشی از سیستم شما بیش از حد پیچیده باشد، روش خاصی برای رفع آن وجود دارد - در مراحل کوچک، قطعات جداگانه را دوباره طراحی کنید. هر اصلاح باید به همان اندازه کوچک باشد که با خیال راحت و بدون پیچیدگی بیشتر بتوانید آن را درست کنید. هنگامی که این فرآیند را طی می کنید، بزرگترین خطر این است که با رفع اشکالات خود احتمالاً پیچیدگی بیشتر میشود. به همین دلیل است که در نهایت بسیاری از بازطراحی ها یا بازنویسی ها با شکست روبرو می شوند - آنها پیچیدگی های بیشتری را نسبت به آنچه که رفع می کنند وارد می کنند، یا در نهایت به همان پیچیدگی سیستم اصلی می شوند.

هر مرحله می‌تواند به اندازه دادن یک نام بهتر به یک متغیر یا فقط افزودن چند نظر به کد گیج کننده باشد. اما بیشتر اوقات، این گام‌ها شامل تقسیم یک قطعه پیچیده به چند قطعه ساده است.

به عنوان مثال، اگر یک فایل طولانی دارید که همه کد شما را در بر دارد، با تقسیم یک قطعه کوچک به یک فایل جداگانه، شروع به بهبود آن کنید. سپس طراحی آن قطعه ریز را بهبود ببخشید. سپس قطعه کوچک دیگری از سیستم را به یک پرونده جدید تقسیم کرده و طراحی آن را بهبود بخشید. همین کار را ادامه دهید و در نهایت با یک سیستم قابل اعتماد، قابل فهم و قابل نگهداری مواجه خواهید شد.

توجه به این نکته مهم است که شما نمی توانید نوشتن ویژگی ها را متوقف کنید و مدت طولانی را صرف طراحی مجدد کنید. قانون تغییر به ما می گوید که محیط پیرامون برنامه شما به طور مداوم در حال تغییر است و بنابراین عملکرد برنامه شما باید سازگار باشد. اگر برای مدت زمان قابل توجهی نتوانید از نظر کاربر سازگار شوید و پیشرفت کنید، خطر از دست دادن پایگاه کاربر و مرگ پروژه شما را تهدید می کنید.

خوشبختانه روشهای مختلفی برای ایجاد تعادل در میان این ویژگی نوشتن و کنترل پیچیدگی وجود دارد. یکی از بهترین راه ها این است که طراحی مجدد خود را صرفاً با هدف تسهیل در اجرای برخی ویژگی های خاص و سپس پیاده سازی آن ویژگی انجام دهید. به این ترتیب، شما مرتب بین کار طراحی دوباره و ویژگی جابجا می شوید.

طراحی مجدد برای یک ویژگی

پروژه ای به نام Bugzilla تمام داده های خود را در یک پایگاه داده ذخیره می کند. Bugzilla فقط از یک سیستم پایگاه داده خاص برای ذخیره اطلاعات به نام OldDB پشتیبانی می کند. برخی از مشتریان جدید می خواهند از یک سیستم پایگاه داده متفاوت برای ذخیره داده ها به نام NewDB استفاده کنند. این مشتریان دلایل خوبی برای خواستن این ویژگی دارند: آنها NewDB را بسیار بهتر از OldDB می فهمند و از قبل NewDB را در شرکت های خود اجرا می کنند. اما همه مشتریان موجود می خواهند از OldDB استفاده کنند.

بنابراین ، Bugzilla باید پشتیبانی از بیش از یک پایگاه داده را شروع کند. این به تغییرات زیادی در کد نیاز دارد ، زیرا Bugzilla هیچ کد متمرکز برای ذخیره و دریافت اطلاعات از پایگاه داده ندارد. در عوض ، بسیاری از دستورات پایگاه داده سفارشی در سراسر کد وجود دارد که مختص OldDB است و در NewDB کار نمی کند.

یک گزینه این است که دستورات if را در کل پایگاه کد استفاده کنید، نوشتن کد های مختلف برای NewDB و OldDB در همه جا که به پایگاه داده دسترسی پیدا می کند. این تقریباً پیچیدگی کل کدبیس را دو چندان می کند و تیم Bugzilla تنها از چند برنامه نویس نیمه وقت تشکیل شده است. اگر پیچیدگی سیستم دو برابر شود ، دیگر نمی توانند آن را حفظ کنند.

در عوض ، تیم Bugzilla تصمیم به طراحی مجدد سیستم می گیرد تا بتواند از چندین پایگاه داده به راحتی پشتیبانی کند. این یک پروژه عظیم است. در اینجا یک نمای کلی از سطح بالایی از چگونگی انجام آن وجود دارد:

برخی دستورات استاندارد پایگاه داده وجود دارد که روی هر سیستم پایگاه داده کار می کنند ، اما همیشه از آنها استفاده نمی شود. سیستم را مرور کرده و هر بار یک پرونده را درست کنید ، آن را تغییر دهید تا در صورت امکان از دستورات استاندارد استفاده کنید.

برای دستورات پایگاه داده که نسخه استانداردی وجود ندارد ، توابعی ایجاد کنید که دستور درستی را برای پایگاه داده مورد استفاده بازگرداند. یک عملکرد برای یک دستور غیر استاندارد ایجاد می شود و سپس هر نمونه از آن دستور غیر استاندارد با فراخوانی عملکرد جایگزین می شود. این روند را ادامه دهید تا تمام عملکردهای غیر استاندارد از بین بروند.

تعداد زیادی کد به طور کامل در اطراف ویژگی هایی طراحی شده اند که فقط در OldDB وجود دارد. استفاده از این ویژگی های خاص OldDB را متوقف کنید و در عوض از ویژگی های استاندارد استفاده کنید که روی همه سیستم های پایگاه داده کار خواهند کرد. در صورت لزوم در چند مرحله این ویژگی ها را یک به یک برطرف کنید.

طراحی مجدد سیستم نصب Bugzilla به گونه‌ای که بتواند خودش را روی هر سیستم پایگاه‌داده راه‌اندازی کند، نه فقط OldDB. این شامل طراحی مجدد سیستم نصب به ساده‌تر و سپس تنظیم آن کد ساده برای پشتیبانی هم OldDB و هم NewDB است.

هر مرحله بالاتر یک پروژه به خودی خود است. همه آنها به مراحل کوچکتر تقسیم می شوند ، بنابراین می توان روی هر کار طراحی خوبی انجام داد. همچنین ، پس از ایجاد هرگونه تغییر ، سیستم مورد آزمایش قرار می گیرد تا مطمئن شوید که در OldDB همچنان به همان روال قبلی عمل می کند.

آیا این منجر به ایجاد یک سیستم کامل می شود؟ خیر ، اما منجر به سیستمی بهتر از گذشته می شود - علاوه بر پشتیبانی از NewDB ، نگهداری از کد اکنون بسیار آسان تر است. سرانجام Bugzilla برای پشتیبانی از چهار سیستم پایگاه داده مختلف گسترش یافت ، همه به این دلیل که این کار پشتیبانی از سیستم های جدید را بسیار آسان تر می کند.

ساده تر کردن یک قطعه

موارد بالا خوب است، اما در واقع برای ساده تر کردن یک قطعه چه کاری انجام می دهید؟ خوب، این جایی است که تمام دانش موجود جهان در مورد طراحی نرم‌افزار به نمایش در می‌آید. این امر به مطالعه در مورد الگوهای طراحی ، روش های مقابله با کد قدیمی و به طور کلی همه ابزارهای مهندسی نرم افزار کمک زیادی می کند. دانستن چندین زبان برنامه نویسی و آشنایی با بسیاری از کتابخانه های مختلف می تواند بسیار مفید باشد، زیرا هر یک شامل روش های مختلف تفکر در مورد مشکلاتی است که می تواند برای شرایط شما قابل اجرا باشد، حتی اگر از آن زبان ها یا کتابخانه ها استفاده نکنید.

مطالعه آن مطالب گزینه های زیادی را برای انتخاب در اختیار شما قرار می دهد در صورتی که با پیچیدگی روبرو هستید. قوانین طراحی نرم افزار می تواند به شما کمک کند گزینه های مناسب را انتخاب کنید و سپس قضاوت و تجربه شما می تواند تعیین کند که واقعاً با مشکل خاص خود چه کاری انجام دهید.

هرگز به صورت ربات وار از یک ابزار صرفا به این دلیل استفاده نکنید زیرا برخی از مقامات آن را بهترین تشخیص داده اند — همیشه کاری که درست است را برای کدی که دارید و وضعیتی که در آن هستید انجام دهید.

با این حال، گاهی ممکن است به یک قطعه کد نگاه کنید و هیچ ابزاری برای ساده کردن آن نداشته باشید. یا ممکن است در برنامه نویسی تازه کار باشید و وقت آن را ندارید که بلافاصله همه این اطلاعات را مطالعه کنید. در این حالت، شما باید فقط به پیچیدگی نگاه کنید و از خودتان بپرسید، "چگونه می توان کار کردن با آن را ساده‌تر و یا بیشتر قابل‌درک کرد؟" این سوال کلیدی در پشت هر ساده‌سازی است . هر پاسخ واقعی به آن روشی معتبر برای ساده سازی کد شماست.

ابزارها و تکنیک های طراحی نرم افزار فقط به ما کمک می کنند تا پاسخ های بهتری پیدا کنیم.

پیچیدگی غیر قابل اصلاح

هنگامی که روی ساده سازی سیستم خود کار می کنید، ممکن است متوجه شوید که جلوگیری از برخی از پیچیدگی ها، مانند پیچیدگی سخت افزار اصلی، دشوار است. اگر با این قبیل با پیچیدگی های غیر قابل اصلاح روبرو شدید ، هدف شما پنهان کردن پیچیدگی است. پوششی اطراف آن قرار دهید که استفاده و درک آن برای سایر برنامه نویسان ساده باشد.

بازنویسی

برخی از طراحان ، وقتی با یک سیستم بسیار پیچیده روبرو می شوند، آن را بیرون می اندازند و از نو شروع می کنند.

با این حال ، بازنویسی یک سیستم از پایه اساساً پذیرفتن شکست به عنوان یک طراح است. این بیان میکند که: ما نتوانستیم یک سیستم قابل نگهداری طراحی کنیم و باید از ابتدا شروع کنیم."

برخی معتقدند که در نهایت همه سیستم ها باید دوباره بازنویسی شوند. این درست نیست. می توان سیستمی را طراحی کرد که هرگز نیازی به دور انداختن نداشته باشد. یک طراح نرم افزار که می گوید "به هر حال باید روزی همه چیز را دور بریزیم" چیزی شبیه یک معمار ساختمان است که می گوید "به هر حال روزی این آسمان خراش سقوط خواهد کرد."

اگر آسمان خراش ضعیف طراحی شده بود و به خوبی نگهداری نمی شد، بله، روزی سقوط می کند. اما اگر از شروع درست ساخته شود و پس از آن به درستی نگهداری شود، چرا باید فرو بریزد؟

ساخت سیستم های نرم افزاری قابل نگهداری همانند ساخت آسمان‌خراش‌های بلند و محکم ممکن است.

اکنون با تمام آنچه گفته شد، شرایطی وجود دارد که بازنویسی در آنها قابل قبول است. با این حال، آنها بسیار نادر هستند. فقط در صورت صحت همه موارد زیر باید دوباره آن را بازنویسی کنید:

1. شما برآورد دقیقی ایجاد کرده اید که نشان می دهد بازنویسی سیستم نسبت به طراحی مجدد سیستم موجود از نظر زمان از بازدهی بیشتری برخوردار است. فقط حدس نزنید – آزمایشات واقعی را با طراحی مجدد سیستم موجود انجام دهید تا ببینید چطور پیش خواهد رفت. مقابله با پیچیدگی موجود و حل بخشی از آن بسیار دشوار است، اما در واقع باید چند بار این کار را انجام دهید تا بدانید که برای رفع مشکل همه آن به چه مقدار تلاش نیاز دارید.
2. شما زمان بسیار زیادی برای صرف یک سیستم جدید اختصاص داده اید.
3. شما به نوعی طراح بهتری نسبت به طراح اصلی سیستم هستید یا اگر طراح اصلی باشید، از زمانی که سیستم اصلی را طراحی کرده اید، مهارت های طراحی شما به شدت بهبود یافته است.
4. شما کاملاً قصد دارید این سیستم جدید را در یک سری مراحل ساده طراحی کنید و کاربرانی داشته باشید که می توانند برای هر مرحله در طول مسیر بازخورد خود را شما ارائه دهند.
5. منابع در دسترس شماست تا هم سیستم موجود را حفظ کنید و هم سیستم جدیدی را همزمان طراحی کنید. هرگز نگهداری سیستمی را که اکنون در حال استفاده است متوقف نکنید تا برنامه نویسان بتوانند آن را دوباره بنویسند.

در صورت استفاده, سیستم ها باید همیشه نگهداری شوند. و به یاد داشته باشید که توجه شخصی شما نیز یک منبع است "که باید در اینجا مورد توجه قرار گیرد اگر قرار است روی هر دو کار کنید آیا هر روز وقت کافی دارید که بتوانید به طور همزمان طراح سیستم جدید و سیستم قدیمی باشید ؟

اگر تمام نکات فوق صحیح باشد ، ممکن است در شرایطی قرار بگیرید که بازنویسی قابل قبول باشد. در غیر این صورت، کار صحیح این است که پیچیدگی سیستم موجود را با بهبود طراحی سیستم در یک سری مراحل ساده بدون بازنویسی کنترل

کنید.

فصل 10

آزمایش کردن

هیچ اطمینانی وجود ندارد که یک برنامه در آینده اجرا شود - تنها این اطمینان وجود دارد که یک برنامه اکنون در حال اجراست. حتی اگر یک‌بار آن را اجرا کنید، ممکن است دوباره اجرا نشود. شاید محیط اطراف آن تغییر کند و دیگر اجرا نشود. شاید شما آن را روی یک کامپیوتر دیگر اجرا کنید، و روی دستگاه جدید کار نکند.

با این وجود، یک امید وجود دارد - ما محکوم به عدم اطمینان بی پایان به عملکرد نرم افزار خود نیستیم. قانون آزمایش راه حل را به ما می گوید:

درجه ای که می دانید نرم افزار شما چگونه رفتار می کند، درجه آزمایش دقیق آن است.

هر چه به تازگی نرم‌افزار خود را آزمایش کرده‌اید، احتمال بیشتری وجود دارد که هنوز هم کار کند. هرچه در محیط های بیشتری آزمایش شده باشد، می توانید اطمینان حاصل کنید که در آن شرایط کار می کند. این بخشی از هدف ماست وقتی که در مورد "درجه" آزمایش صحبت می کنیم - چند جنبه از نرم افزار را آزمایش کرده اید؟ چند وقت اخیر؟ و در چند محیط مختلف؟ - به طور کلی ، می توانید به سادگی بگویید:

تا زمانی که آن را امتحان نکرده باشید ، نمی دانید که موثر است.

اگرچه بگویید "کار می کند" کاملا مبهم است - منظورتان از "کارها" چیست؟ آنچه در هنگام آزمایش می دانید این است که نرم افزار شما همانگونه که قصد داشتید رفتار می کند. بنابراین شما باید بدانید چه رفتاری در نظر گرفته‌اید. این ممکن است احمقانه و واضح به نظر برسد، اما این یک واقعیت مهم در آزمایش است.

شما باید با هر آزمایش یک سوال بسیار دقیق بپرسید و پاسخ خیلی خاصی به دست آورید. این سوال می تواند چنین باشد: "چه اتفاقی می افتد زمانی که کاربر پس از شروع برنامه این دکمه را فشار دهد؟، وقتی که برنامه قبلاً هرگز شروع نشده است، " و شما باید به دنبال پاسخ خاصی باشید ، مانند: "برنامه پنجره ای را نشان می دهد که می گوید:" سلام جهان! "

بنابراین، شما یک سوال دارید، و می‌دانید که پاسخ باید چه باشد. اگر پاسخ دیگری می‌گیرید، نرم‌افزار شما "کار نمی‌کند".

گاهی اوقات آزمایش یک رفتار بسیار دشوار است و شما فقط می توانید بپرسید، "اگر کاربری این کار را انجام دهد ، آیا برنامه خراب می شود؟" و انتظار پاسخ "نه" را داشته باشید اما با استفاده از نرم افزارهای خوب طراحی شده، در بیشتر مواقع، می توانید اطلاعات بسیار دقیق تری از این آزمایشات را بدست آورید.

☆همچنین باید آزمایشات خود را دقیق انجام دهید. اگر آنها به شما بگویند که برنامه در هنگام عدم عملکرد خود به درستی رفتار می کند - یا به شما می گویند که وقتی واقعاً خوب کار می کند خراب است - این تست ها نادرست هستند.

در آخر ، باید نتایج آزمون های خود را مشاهده کنید تا معتبر باشند. در صورت عدم موفقیت، باید راهی وجود داشته باشد تا بدانید که آنها شکست خورده اند، و به طور دقیق آنها چگونه شکست خورده اند.

نادیده گرفتن آزمایش می تواند آسان باشد. ما برخی از کدها را می نویسیم، ذخیره می کنیم و فراموش می کنیم که آیا هرگز واقعاً کار می کند یا نه. اما مهم نیست که شما چقدر برنامه نویس خوبی باشید، مهم نیست که چند اثبات برای نشان دادن درست بودن کد خود ارائه دهید، هرگز نخواهید فهمید که کار می کند یا نه مگر اینکه واقعاً سعی در استفاده از آن داشته باشید. و اگر در هر مرحله بخشی از نرم افزار خود را تغییر دهید، دیگر نمی دانید که آن قطعه کار می کند. باید دوباره آزمایش شود. بعلاوه ، آن قطعه احتمالاً به تعداد زیادی قطعه دیگر متصل است ، بنابراین اکنون نمی دانید که هر کدام از این قطعات کار می کنند یا خیر. اگر تغییر شما به اندازه کافی بزرگ باشد ، ممکن است مجبور شوید دوباره کل برنامه را آزمایش کنید.

بدیهی است ، شما نمی خواهید هر بار که یک تغییر کوچک ایجاد می کنید، کل برنامه خود را آزمایش کنید. بنابراین، در دوران مدرن، توسعه دهندگان معمولاً این قانون را با ایجاد تست های خودکار برای هر قطعه کدی که می نویسند، انجام میدهند. نکته خوب این است که آنها میتوانند بلافاصله پس از ایجاد هرگونه تغییری تست ها را اجرا کنند و این آزمایشات خودکار تک تک قطعات سیستم را آزمایش می کنند تا مطمئن شوند بعد از هر تغییر همه کارها همچنان ادامه دارد.

فصل 11

پیوست اول

قوانین طراحی نرم افزار

این پیوست خلاصه همه قوانین واقعی مورد بحث در این کتاب است :

1. هدف از نرم افزار کمک به مردم است.

2. معادله طراحی نرم افزار:

$$D = \frac{V_n + V_f}{E_i + E_m}$$

D

به معنای مطلوبیت تغییر است.

V_n

به معنای ارزش حال حاضر است.

V_f

به معنای ارزش آینده است.

E_i

به معنای تلاش برای اجرا است.

E_m

مخفف تلاش برای نگهداری است.

این قانون اولیه طراحی نرم افزار است. با گذشت زمان ، این معادله به موارد زیر کاهش می یابد:

$$D = \frac{V_f}{E_m}$$

که نشان می‌دهد که کاهش تلاش برای تعمیر و نگهداری مهم‌تر از آن است که تلاش برای اجرا را کاهش دهیم.

۳. قانون تغییر: هرچه برنامه شما به مدت طولانی‌تری وجود داشته باشد، احتمال بیشتری وجود دارد که هر قطعه از آن تغییر کند.

1. قانون احتمال عیب و نقص: احتمال ورود عیب و نقص در برنامه شما متناسب با اندازه تغییراتی است که در آن ایجاد می کنید.

2. قانون سادگی: سهولت در نگهداری هر نرم افزار متناسب با سادگی قطعات جداگانه آن است.

3. قانون آزمایش: درجه‌ای که شما می‌دانید نرم‌افزار شما چگونه رفتار می‌کند درجه‌ای است که شما دقیقا آن را امتحان کرده‌اید.

خودشه. حقایق و ایده های بسیاری در این کتاب مورد بحث قرار گرفت ، اما این شش مورد قوانین طراحی نرم افزار است. توجه داشته باشید که از همه این موارد ، مهمترین نکته ای که باید بخاطر بسپارید هدف نرم افزار است که شکل کاهش یافته معادله طراحی نرم افزار و قانون سادگی است.

اگر می خواهید مهمترین حقایقی را که باید در مورد طراحی نرم افزار در ذهن داشته باشید در دو جمله ساده خلاصه کنید، این موارد عبارتند از:

• کاهش تلاش برای تعمیر و نگهداری مهم‌تر از آن است که تلاش برای اجرا را کاهش دهد.

• تلاش برای نگهداری متناسب با پیچیدگی سیستم است.

تنها با داشتن این دو گزاره و درک هدف از نرم افزار ، شما می توانید کل علم طراحی نرم افزار را مجدداً متحول کنید ، به شرطی که بدانید پیچیدگی سیستم در واقع از پیچیدگی قطعات جداگانه آن ناشی می شود.

فصل 12

پیوست دوم

حقایق، قوانین و تعاریف

این پیوست هر واقعیت مهم ، قانون ، قاعده و تعریفی است که در این کتاب آورده شده است را لیست میکند:

- واقعیت: تفاوت بین یک برنامه‌نویس بد و یک برنامه‌نویس خوب درک است. به همین دلیل است که برنامه نویسان بد کاری را که انجام می‌دهند درک نمی‌کنند ولی برنامه نویسان خوب درک می‌کنند.
- قاعده: یک "برنامه نویس خوب" باید تمام تلاش خود را انجام دهد تا آنچه را که می نویسد برای برنامه نویسان دیگر ساده کند.
- تعریف: یک برنامه عبارت است از:
  - توالی دستورالعمل های داده شده به کامپیوتر
  - اقداماتی که توسط کامپیوتر در نتیجه دستورالعمل داده شده انجام می شود
- تعریف: هر چیزی که شامل معماری سیستم نرم افزاری شما یا تصمیمات فنی شما هنگام ایجاد سیستم باشد ، در گروه "طراحی نرم افزار" قرار می گیرد.
- واقعیت : هرکسی که نرم افزار می نویسد یک طراح است.
- قاعده : طراحی دموکراتیک نیست. تصمیمات باید توسط افراد گرفته شود.
- واقعیت : قوانینی در زمینه طراحی نرم افزار وجود دارد ، می توان آنها را شناخت و می توانید بدانید آنها آنها ابدی ، تغییرناپذیر و اساساً واقعی هستند و موثرند.
- **قانون** : هدف از نرم افزار کمک به مردم است.
- واقعیت: اهداف طراحی نرم افزار عبارتند از:
  - به ما اجازه دهد تا نرم افزاری را بنویسیم که تا حد امکان مفید باشد
  - به نرم‌افزار ما اجازه دهد تا جایی که ممکن است به مفید بودن خود ادامه دهد
  - برای طراحی سیستم هایی که بتوانند به آسانی توسط برنامه نویسان خود ایجاد و نگهداری شوند ، تا بتوانند مفید باشند

• **قانون** : معادله طراحی نرم افزار :

$$D = \frac{V_n + V_f}{E_i + E_m}$$

این قانون اولیه طراحی نرم افزار است. یا به فارسی:

مطلوبیت یک تغییر مستقیماً با ارزش اکنون به علاوه ارزش آینده متناسب است و به صورت معکوس با تلاش برای اجرا به علاوه تلاش برای نگهداری متناسب است.

با گذشت زمان ، این معادله به موارد زیر کاهش می یابد:

$$D = \frac{V_f}{E_m}$$

که نشان می‌دهد که کاهش تلاش برای تعمیر و نگهداری مهم‌تر از آن است که تلاش برای اجرا را کاهش دهیم.

- قاعده: سطح کیفیت طراحی شما باید متناسب با مدت زمان آینده باشد که در آن سیستم شما به مردم کمک خواهد کرد.
- قاعده: مواردی درباره آینده وجود دارد که شما نمی دانید.
- واقعیت: رایج ترین و فاجعه بارترین خطایی که برنامه نویسان مرتکب می شوند پیش بینی چیزی در مورد آینده است در حالی که در واقع نمی توانند از آن مطلع شوند.
- قاعده: اگر سعی در پیش بینی آینده نداشته باشید ،در ایمن ترین حالت قرار دارید در عوض تمام تصمیمات طراحی خود را بر اساس اطلاعات مشخص زمان حال اتخاذ کنید.
- **قانون** : قانون تغییر: هرچه برنامه شما به مدت طولانی‌تری وجود داشته باشد، احتمال بیشتری وجود دارد که هر قطعه از آن تغییر کند.
- واقعیت: سه اشتباهی (که به آنها "سه نقص" در این کتاب گفته می شود) که طراحان نرم افزار در کنار آمدن با قانون تغییر مستعد انجام آن هستند:
  - نوشتن کدی که مورد نیاز نیست
  - آسان نکردن کد برای تغییر
  - بیش از حد عمومی بودن
- قاعده: کد را تا زمانی که واقعاً به آن احتیاج ندارید ، ننویسید و کدی را که استفاده نمی شود حذف کنید.
- قاعده: کد باید بر اساس آنچه اکنون می دانید طراحی شود ، نه بر اساس آنچه فکر می کنید در آینده اتفاق خواهد افتاد.
- واقعیت: وقتی طراحی شما به جای ساده کردن کارها ، کارها را پیچیده تر می کند ، شما بدرحال مهندسی بیش از حد هستید

- قاعده: فقط به همان اندازه عمومی باشید که می دانید در حال حاضر لازم است.
- قاعده: با انجام تدریجی توسعه و طراحی می توانید از سه نقص جلوگیری کنید.
- **قانون** : قانون احتمال عیب و نقص: احتمال ورود عیب و نقص به برنامه شما متناسب با اندازه تغییراتی است که در آن ایجاد می کنید.
- قاعده: بهترین طراحی آن است که امکان ایجاد بیشترین تغییر در محیط با کمترین تغییر در نرم افزار را فراهم کند.
- قاعده: هرگز چیزی را "تعمیر" نکنید ، مگر اینکه مشکلی باشد و شواهدی دارید که نشان می دهد مشکل واقعاً وجود دارد.
- قاعده: در هر سیستم خاصی ، هرگونه اطلاعات ، در حالت ایده آل ، فقط یک بار باید وجود داشته باشد.
- **قانون** : قانون سادگی: سهولت در نگهداری هر نرم افزار متناسب با سادگی قطعات جداگانه آن است.
- واقعیت: سادگی نسبی است.
- قاعده: اگر واقعاً میخواهید موفق شوید ، بهتر است به صورت احمقانه ای ساده باشید.
- قاعده: ثابت قدم باشید.
- قاعده: خوانایی کد در درجه اول به چگونگی اشغال فضا توسط حروف و نمادها بستگی دارد.
- قاعده: اسامی باید آنقدر طولانی باشند که بتوانند آنچه را که وجود دارد، به طور کامل منتقل کنند بدون اینکه آن قدر طول باشند که خواندن آنها دشوار شود.
- قاعده: نظرات باید توضیح دهند که چرا کد کاری را انجام می دهد ، نه کاری را که انجام می دهد.
- قاعده: سادگی به طراحی نیاز دارد.
- قاعده: با استفاده از موارد زیر می توانید پیچیدگی ایجاد کنید:
    - گسترش هدف نرم افزار خود
    - افزودن برنامه نویس به تیم
    - تغییر چیزهایی که نیازی به تغییر ندارند
    - محصور بودن در فن آوری های بد
    - سوء تفاهم

- طراحی ضعیف یا عدم طراحی
- اختراع دوباره چرخ
- نقض هدف نرم افزار خود

- قاعده: با بررسی پتانسیل بقا ، قابلیت همکاری و توجه به کیفیت می توانید بد بودن یا نبودن یک فناوری را تعیین کنید.
- قانون: غالباً ، اگر مسئله بسیار پیچیده باشد ، این بدان معنی است که در طراحی جایی در زیر سطح پیچیدگی خطایی رخ داده است.
- قاعده: وقتی با پیچیدگی روبرو شدید ، بپرسید ، "چه مشکلی را می خواهید حل کنید؟"
- قاعده: دشوارترین مشکلات طراحی را می توان با نقاشی ساده یا نوشتن روی کاغذ حل کرد.
- قاعده: برای رسیدگی به پیچیدگی های سیستم خود ، قطعات جداگانه را در مراحل کوچک از نو طراحی کنید.
- واقعیت: سوال کلیدی پشت سر همه ساده‌سازی معتبر این است که "چگونه می توان این کار را ساده‌تر و یا بیشتر قابل‌درک کرد؟"
- قاعده: اگر در خارج از برنامه خود با پیچیدگی غیر قابل اصلاح روبرو هستید ، پوششی در اطراف آن قرار دهید که برای برنامه نویسان دیگر ساده است.
- قاعده: بازنویسی فقط در یک گروه بسیار محدود قابل قبول است.
- **قانون** : قانون آزمایش: درجه‌ای که شما می‌دانید نرم‌افزار شما چگونه رفتار می‌کند درجه‌ای است که شما دقیقا آن را امتحان کرده‌اید.
- قاعده: تا زمانی که آن را امتحان نکرده باشید ، نمی دانید که موثر است.

درباره نویسنده

Max Kanat-Alexander، معمار ارشد پروژه متن باز Bugzilla، مهندس نرم افزار Google و نویسنده، از هشت سالگی کامپیوترها را تعمیر می کرده و از چهارده سالگی نرم افزار می نویسد. او نویسنده codesmplicity.com و fedorafaq.org است و در حال حاضر در شمال کالیفرنیا زندگی می کند.